2011-06

A MAC layer covert channel in 802.11 networks

Gonçalves, Ricardo André Santana

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A MAC LAYER COVERT CHANNEL IN 802.11 NETWORKS

by

Ricardo André Santana Gonçalves

June 2011

Thesis Co-Advisors: Murali Tummala
John McEachen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2011	Master's Thesis

4. TITLE AND SUBTITLE A MAC Layer Covert Channel in 802.11 Networks	5. FUNDING NUMBERS
6. AUTHOR(S) Ricardo André Santana Gonçalves	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	A

13. ABSTRACT (maximum 200 words)

Covert channels in modern communication networks are a source of security concerns. Such channels can be used to facilitate command and control of botnets or inject malicious contents into unsuspected end-user devices or network nodes. The vast majority of the documented covert channels make use of the upper layers of the Open Systems Interconnection (OSI) model. In this thesis, we present a new covert channel in IEEE 802.11 networks, making use of the Protocol Version field in the Medium Access Control (MAC) header. This is achieved by forging modified Clear To Send (CTS) and Acknowledgment (ACK) frames. Forward error correction mechanisms and interleaving were implemented to increase the proposed channel's robustness to error. A laboratory implementation of the proposed channel is presented by developing the necessary code in Python, operating in a Linux environment. We present the results of tests conducted on the proposed channel, including measurements of channel errors, available data rate for transmission, and level of covertness.

14. SUBJECT TERMS	15. NUMBER OF PAGES
IEEE802.11 MAC frame, Frame forging, Covert channel, Protocol version	103
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A MAC LAYER COVERT CHANNEL IN 802.11 NETWORKS

Ricardo André Santana Gonçalves
Lieutenant, Portuguese Navy
B.S., Portuguese Naval Academy, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2011

Author: Ricardo André Santana Gonçalves

Approved by: Murali Tummala
Thesis Co-Advisor

John McEachen
Thesis Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Covert channels in modern communication networks are a source of security concerns. Such channels can be used to facilitate command and control of botnets or inject malicious contents into unsuspected end-user devices or network nodes. The vast majority of the documented covert channels make use of the upper layers of the Open Systems Interconnection (OSI) model. In this thesis, we present a new covert channel in IEEE 802.11 networks, making use of the Protocol Version field in the Medium Access Control (MAC) header. This is achieved by forging modified Clear To Send (CTS) and Acknowledgment (ACK) frames. Forward error correction mechanisms and interleaving were implemented to increase the proposed channel's robustness to error. A laboratory implementation of the proposed channel is presented by developing the necessary code in Python, operating in a Linux environment. We present the results of tests conducted on the proposed channel, including measurements of channel errors, available data rate for transmission, and level of covertness.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ACK	Acknowledgment
AP	Access Point
BSS	Basic Service Set
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
CTS	Clear To Send
DNS	Domain Name System
DoD	Department of Defense
DoS	Denial of Service
DS	Distribution System
FC	Frame Control
FCS	Frame Check Sequence
FEC	Forward Error Correction
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPS	Intrusion Prevention System
LLC	Logical Link Control
MAC	Medium Access Control
OS	Operating System
OSI	Open Systems Interconnection
PV	Protocol Version

QoS	Quality of Service
RTS	Request To Send
SNR	Signal-To-Noise Ratio
TCSEC	Trusted Computer System Evaluation Criteria
WEP	Wired Equivalent Privacy

ACKNOWLEDGMENTS

Quero agradecer aos meus pais por terem plantado em mim o bichicho de querer saber e por inspirarem a coragem necessária para enfrentar os desafios que a vida nos coloca. Deixo também um agradecimento muito especial à minha mulher, Inês, e às nossas duas pestes, Maria e Rita, por todo o amor, paciência, apoio e compreensão em todos os momentos, independentemente das horas a que chegasse a casa. Uma última palavra de agradecimento à Marinha de Guerra Portuguesa e a todos os profissionais que tornaram esta experiência possível.

I would like to thank my thesis advisor, Professor Murali Tummala, for his infinite patience and excellent guidance during this learning experience. I have much to thank him and his contribution to the successful completion of this thesis. I would like to thank Professor John McEachen for his time and insightful suggestions of this thesis and valuable feedback.

Last but not least, I would like to extend my appreciation to all who have contributed to the completion of this work in one way or another.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

IEEE 802.11, also known as WiFi, is one of the most widely used set of standards in today's wireless network communications. It is present in a wide variety of electronic equipment, from smartphones and laptops to kitchen appliances and automobiles. According to an industry report, in 2012 over one billion devices will be shipped with technology based on this standard onboard, and the number is projected to be over two billion in 2014. The technical capabilities and the mobility provided to the user make it one of the most successful wireless networking systems.

As in any other type of network communication standard, security plays a key role. Mobility and ease of access are attractive characteristics to the end users, but along with them come additional security concerns. It is important to evaluate the possible weaknesses and vulnerabilities of a standard in order to determine relevant security challenges. The particular focus in this work is covert channels, which have the characteristic of being hard to detect unless we know in advance what we are looking for.

Covert channels come up as one of many aspects involved in the security evaluation of a standard and can pose a threat to the unaware user. A covert channel is a method to transmit information using the communication protocol in a way that was not intended or anticipated by

the developers. Such covert channels can be used to exfiltrate information from the user's device, propagate malware or control a botnet.

The objectives of this thesis were to identify, implement and test a proof-of-concept covert channel in IEEE 802.11 networks. This was achieved by forging control frames and exploiting specific bits in the Medium Access Control (MAC) header.

In this thesis, we developed the necessary code to implement the proposed covert channel, conducted laboratory experiments, and measured and analyzed the results. Operational IEEE 802.11 networks were monitored prior to designing the proposed covert channel, allowing us to gather enough information to make a sound decision on which frames to forge and how to manipulate them. In order to improve the error performance of the proposed covert channel, several techniques were used, such as convolutional coding and bit interleaving. Detectability and mitigation techniques were also addressed, as well as a throughput analysis.

Ideas for future work include optimization of the channel throughput, increasing the channel's robustness to errors and exploring the proposed covert channel concept in other emerging wireless standards, such as IEEE 802.16 (WiMAX) or Long Term Evolution (LTE).

I. INTRODUCTION

As wireless networks become more ubiquitous, so do our dependencies on them. In a relatively short period, day-to-day use of wireless networks and mobile devices have become a large part of our modern-day lives. This trend is likely to continue in the coming years, regardless of the specific technologies. Mobility and ease of access are very attractive characteristics to the end users, but along with them come additional security concerns [1,2].

One security-related issue associated with communication networks, wired or wireless, is the concept of a covert channel, which takes advantage of the very fabric of communication networks and exploits them in a way that allows the communication protocols to become the unintended carrier of messages. This idea of network covert channels was documented 25 years ago by Girling [3]. However, the concept of a system-based covert channel was initially presented by Lampson in 1973 [4]. Extensive progress has been made in protocol design since then, but covert channels are still a security concern. It becomes difficult to account for the existence of every possible variation of these channels.

According to the Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC), a covert channel is defined as "any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy [5]." This means a protocol may be used in a way that was not intended or anticipated by the designers.

As networks and respective protocols have evolved and changed, so have the documented covert channels. A search for possible covert channels begins every time a new protocol is implemented or an existing one is modified.

A new covert channel embedded in the Medium Access Control (MAC) layer of an IEEE 802.11-2007 [6] wireless network is presented and explored in this thesis.

A. MOTIVATION

Over the last decade, wireless communications has played a key role in user mobility, a much appreciated benefit of such technology. On the other hand, wireless access and user mobility pose security challenges due to underlying vulnerabilities associated with covert channels and other weaknesses.

In order to protect wireless networks from being exploited, we need to constantly evaluate their vulnerabilities and devise techniques to mitigate them. Finding possible covert channels presents an ongoing challenge, and the possible uses for such channels range from well-intentioned authentication mechanisms [7], to malware propagation [8], exfiltration [8,9] or command and control of botnets [10].

The above gives us enough reason to ask ourselves, What can be worse than not being able to decipher the contents of an unwanted communication? Our answer would be not knowing such a communication is even taking place. The power contained in covert channels is that they have the possibility of being in operation long before they are detected and identified as channels. The ability to

communicate in this manner gives the user who knows of the covert channel a tool that could be used in either a benign or malicious manner.

Although most networks today are protected by intrusion detection systems (IDS) and/or intrusion prevention systems (IPS), an undocumented covert channel can be in operation without triggering an alarm [8]. The key factor is that these covert channels are being operated in the background, making it extremely hard to protect a network against an unknown covert channel. The importance of investigating as many covert channels as possible should be obvious, as each networking standard has its own unique characteristics to exploit. For this reason, it is generally accepted that covert channels cannot be completely eliminated because of numerous variations in their implementation [11,12].

B. OBJECTIVE

The objective of this thesis is to identify, implement and test a proof-of-concept covert channel in an IEEE 802.11-2007 network environment. The purposed covert channel will use the MAC header of control frames to hide the covert information. This will be achieved by forging frames that use the protocol version bits in a way that was not intended by the designers of the IEEE 802.11 standard.

The proposed channel will be implemented using the Python [13] programming language in a Linux environment. A graphical user interface (GUI) that resembles a typical chat room window will be used. Tests will be conducted over an operational network under different conditions. Matlab will be used for analyzing the measurements from the tests.

To increase the proposed channel's robustness to errors, forward error correction and bit interleaving techniques will be used.

C. RELATED WORK

Many covert channels have been documented over the years and reflect the technological stage of the networks at which they were documented. As networking technologies evolve, so do the corresponding protocols and their complexity. With the release of each new networking standard, such complexity opens the door for new covert channels, which makes the research in covert channels challenging.

The vast majority of academic research has focused on documenting covert channels in layer 3 or above of the Open Systems Interconnection (OSI) model, partly neglecting layers 1 and 2 [14]. These types of covert channels based on higher layer protocols span a wider variety of networks, since they are not limited by the physical or medium access mechanisms. The two most explored protocols above layer 2 are IP and TCP [12,15,16]. Even higher layer protocols, such as Internet Control Message Protocol (ICMP), HyperText Transfer Protocol (HTTP) or Domain Name System (DNS), have several documented covert channels [14,17,18].

More recently, researchers began investigating wireless networks, specifically identifying covert channels in the MAC layer [19,20,21,22]. Frame forging plays a key role in this type of covert channel. Creating fake frames with modified header bits is a recurring theme to implement such channels. MAC header fields such as the sequence

number [21], initialization vector [21] or destination address [22], have been used to hide the covert information.

Our work differs from the techniques reported in the literature. In the proposed covert channel, a different MAC header field is used: the protocol version field. Our work also addresses the error robustness and throughput analysis, supported by extensive experimental results.

D. ORGANIZATION

An overview of the IEEE 802.11-2007 standard is presented in Chapter II. The 802.11 data link layer and the different types of MAC frames are discussed. An overview of covert channels and a formal classification are provided. In Chapter III, the formulation and design of the proposed covert channel based on empirical foundations are presented. A large volume of data is collected and analyzed in support of the covert channel formulation.

An implementation of the proposed channel is presented in Chapter IV. We describe the code along with the different test scenarios and experimental setups. An exhaustive analysis of the results is contained in this chapter, including the measurements of channel errors, available throughput, and the level of covertness. Use of forward error correction (FEC) and interleaving to improve the channel performance are discussed.

Chapter V includes conclusions and recommendations for future work. The required steps to launch the proposed covert channel are detailed in Appendix A, and the Python code used to implement it is given in Appendix B.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

In this chapter, we present the basic IEEE 802.11-2007 architectures, types of frames of interest, and an overview of covert channels.

A. OVERVIEW OF IEEE 802.11

1. Protocol Architecture

The IEEE 802.11-based wireless nodes share a common medium for communication. The 802.11 protocol architecture can be seen in Figure 1. It addresses the user access at layers 1 and 2 of the OSI model, i.e., the physical and the data link layer, respectively.

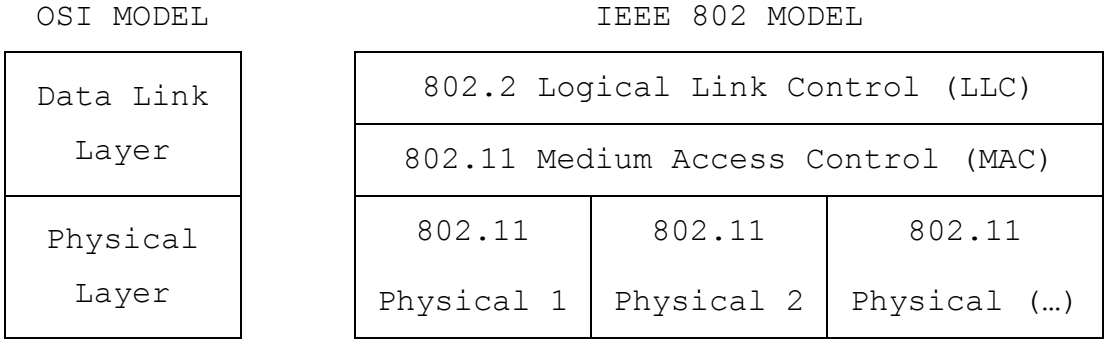


Figure 1. OSI model compared to IEEE 802 protocol architecture.

The logical link control (LLC) acts as a uniform interface between the upper layer and the MAC layer. This enables the network layer to operate normally regardless of the type of MAC being implemented, i.e., for the same LLC, different MAC options are possible.

The medium access control enables the use of a shared medium among several stations. Following the same concept

as before, we see that for the same MAC, different variations of the physical layer can be used. In order to regulate the access to the physical layer, 802.11 makes use of the carrier sense multiple access with collision avoidance (CSMA/CA) scheme [6]. This scheme was developed to avoid collisions due to simultaneous transmissions. Such collisions cause frame loss, reduce the network's throughput and increase delay.

The physical layer enables the transmission of information in the form of electromagnetic signals through the use of different modulation schemes, frequency spreading techniques, multiplexing, etc. As we can see from Figure 1, different types of physical layer technologies were incorporated into the 802.11 standard.

It is important to mention that different versions of the 802.11 protocol are available. These versions differ mainly in the physical layer. The frequency band and bit rate differences among the most common versions are summarized in Table 1. Note that 802.11a, b and g are incorporated in the IEEE 802.11-2007 version of the standard [6]. The 802.11b version is the subject of all the tests and results presented in this thesis.

Table 1. 802.11 versions comparison [From 23].

802.11 version	Frequency Band (Ghz)	Maximum bit rate (Mbps)
802.11-1997	2.4	2
802.11a	5	54
802.11b	2.4	11
802.11g	2.4	54
802.11n	2.4 and/or 5	600

2. Network Architecture

The fundamental building block of the 802.11 architecture is called the basic service set (BSS). One BSS may be connected to other BSSs via a distribution system (DS). Within this framework, stations can connect in ad-hoc mode or infrastructure mode. The simpler case is ad-hoc mode, where two stations can connect directly, point to point, without a DS and an access point (AP). Although convenient, the ad-hoc mode of operation does not support some functions, such as power save.

If we have the stations connecting via an AP and making use of a DS, then we say they are setup in infrastructure mode. A wider range of functions and control mechanisms are possible in this mode, along with centralized security management and extended reach. This type of setup is adequate when we want our wireless network to connect to an existing Ethernet network or other wireless networks in the vicinity, making use of the AP's wider range. An example of such a setup can be seen in

Figure 2. Here, we can see BSS1 being actively protected by a firewall and passively monitored by an Intrusion Detection System (IDS).

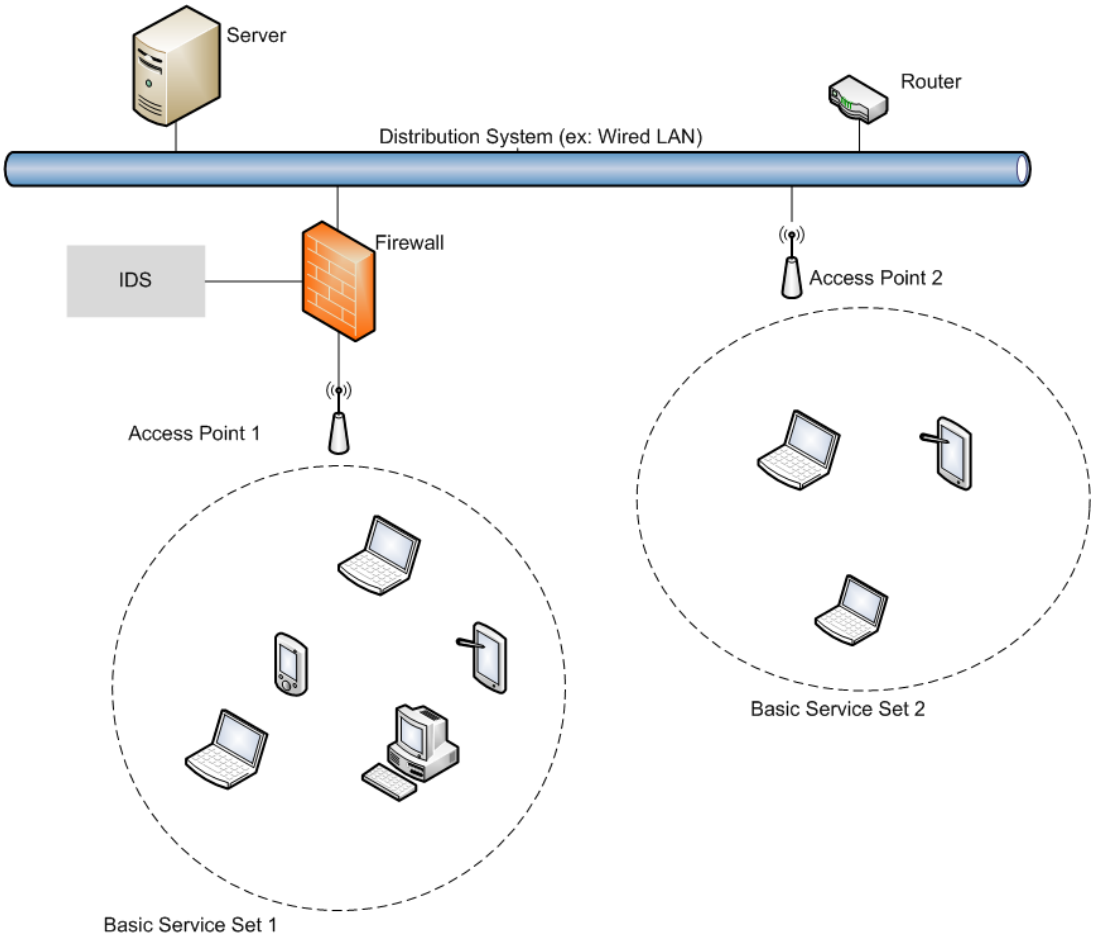


Figure 2. Example of an 802.11 network in infrastructure mode.

B. 802.11 MAC FRAME

1. Header Format

In Figure 3, we can see the generic MAC format for an 802.11 MAC frame. The frame consists of the MAC header, the frame body and the frame check sequence (FCS).

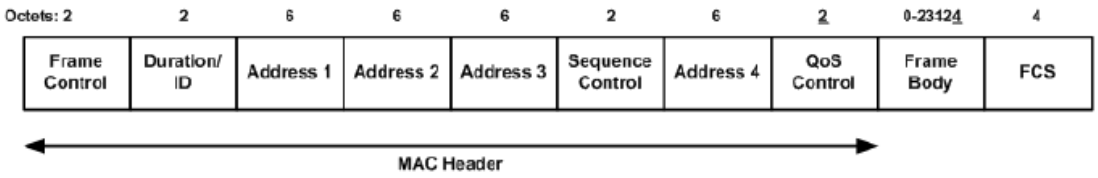


Figure 3. MAC frame format [From 6].

The first field in the MAC header is the frame control (FC), and consists of two octets. In order to better understand the contents and use of this field, a detailed view is depicted in Figure 4.

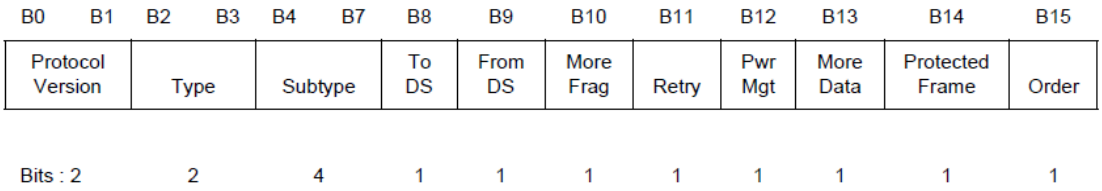


Figure 4. Frame control field [From 6].

Within the FC, the field in which we are interested is the first field, corresponding to the Protocol Version (PV). The PV field consists of two bits that specify the version number of the 802.11 protocol being used. As of this writing, PV is expected to be set to zero [6]. This value may change in the future if a newer version of the standard is released.

The protocol version is the field we will be using for the proposed covert channel. We utilize the remaining three possible combinations of the PV field to hide the covert information.

2. Frame Types of Interest

Four types of frames exist in the 802.11 protocol, as listed in Table 2. We have the management, data, reserved and control frames.

The management frames exist to initiate, establish and maintain the communication between stations. Examples of management frames can be seen in the subtype column of Table 2. Frames responsible for association, disassociation, authentication and beaconing are part of this type. These frames are not very common and for that reason not very interesting for our research.

Data frames are the ones that carry the information and can also provide some services, such as quality of service (QoS).

The reserved frames have no specific task, they are just a type of frames not currently assigned by the standard to perform a specific task.

The last type of frame is the control type. These facilitate the exchange of data frames between stations. Within the existing control subtypes, we are interested in the smaller sized frames, the acknowledgement (ACK) and the clear to send (CTS).

a. Clear to Send/ Request to Send

The IEEE 802.11 MAC layer makes use of the CSMA/CA scheme in order to minimize the number of collisions and subsequent frame loss. This is a way to force the transmitting station to sense the medium, hold its transmissions until the medium is free, and transmit if the media is not in use.

Sometimes, a transmitting station may not be in range of another transmitting station and might sense the medium as free when in fact the medium is being used. A third station, in range of the previous two, will receive both signals simultaneously, sensing a collision. This is known as the hidden node problem [24]. To address this issue, a RTS/CTS handshake mechanism is used. This is done every time a station has information to transmit, making these kind of frames very common.

The CTS is a 14-byte long frame, whereas the RTS is 20 bytes long.

b. Acknowledgment

This type of frame is generated when a station correctly receives a packet, and it is intended to signal the source station that the reception was successful. For this reason, this type of frame also tends to be very common in an operational wireless network. The length of this frame is the same as the CTS, 14 bytes.

The format of the CTS frame, as well as the ACK frame, is shown in Figure 5. Both frames share the same format and only differ in one bit in the subtype field within the frame control, as seen in Table 2. The ACK frame has the subtype value set to 1101; the CTS sets it to 1100.

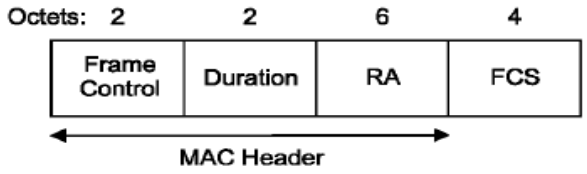


Figure 5. ACK and CTS frame format [From 6].

Table 2. Type and subtype assignments [From 6].

Type value b3 b2	Type description	Subtype value b7 b6 b5 b4	Subtype description
00	Management	0000	Association request
00	Management	0001	Association response
00	Management	0010	Reassociation request
00	Management	0011	Reassociation response
00	Management	0100	Probe request
00	Management	0101	Probe response
00	Management	0110–0111	Reserved
00	Management	1000	Beacon
00	Management	1001	ATIM
00	Management	1010	Disassociation
00	Management	1011	Authentication
00	Management	1100	Deauthentication
00	Management	1101	Action
00	Management	1110–1111	Reserved
01	Control	0000–0111	Reserved
01	Control	1000	Block Ack Request (BlockAckReq)
01	Control	1001	Block Ack (BlockAck)
01	Control	1010	PS-Poll
01	Control	1011	RTS
01	Control	1100	CTS
01	Control	1101	ACK
01	Control	1110	CF-End
01	Control	1111	CF-End + CF-Ack
10	Data	0000	Data
10	Data	0001	Data + CF-Ack
10	Data	0010	Data + CF-Poll
10	Data	0011	Data + CF-Ack + CF-Poll
10	Data	0100	Null (no data)
10	Data	0101	CF-Ack (no data)
10	Data	0110	CF-Poll (no data)
10	Data	0111	CF-Ack + CF-Poll (no data)
10	Data	1000	QoS Data
10	Data	1001	QoS Data + CF-Ack
10	Data	1010	QoS Data + CF-Poll
10	Data	1011	QoS Data + CF-Ack + CF-Poll
10	Data	1100	QoS Null (no data)
10	Data	1101	Reserved
10	Data	1110	QoS CF-Poll (no data)
10	Data	1111	QoS CF-Ack + CF-Poll (no data)
11	Reserved	0000–1111	Reserved

We will focus on ACK and CTS frames, since they are small in size and tend to be large in volume. As a result, if we use them for covert communications, it is difficult to be noticed by monitoring devices such as firewalls and IDSs. The volume of these frames is experimentally verified in Chapter III.

C. COVERT CHANNELS AND RELATED WORK

We presented a definition of covert channels in Chapter I, and we now will look at the different types of channels reported in the literature.

1. Types of Covert Channels

In his 1987 paper, Girling [3] identified two major types of covert channels: storage and timing. The storage covert channels make use of protocols or other mechanisms to write additional information in a way that was not intended, whereas the timing channels signal information between processes by means of varying delays and changing the timing of events [5]. The first type tends to be easier to implement and is the most common. Based on this, the proposed channel in this thesis is a storage channel.

2. Related Work

The work of Frikha et al. [21] was the starting point for this project, inspiring the proposed covert channel configuration. In Frikha's paper, two implementations of a covert channel are presented, both using fields in the 802.11 MAC header. The first uses the eight most significant bits of the sequence control field. This field has a length of two bytes, which is subdivided into two

subfields. The first subfield is the sequence number and comprises the first 12 bits. The following four bits represent the fragment number. By using the eight most significant bits of the sequence number subfield, their covert channel achieves a throughput of one byte per frame.

The second implementation in [21] applies to networks where Wired Equivalent Privacy (WEP) is in use. If this is the case, the initialization vector subfield is used to carry the covert message. This technique allows a throughput of three bytes per frame.

Another covert channel proposed by Butti [22] uses a part of the destination address field of ACK frames to hide the payload. A throughput of one byte per frame is achieved in this case. Butti [22] also presents complete code for the channel's implementation.

Each of these approaches relies on the forging of frames by manipulating the contents of the MAC header in order to hide the covert information.

D. CONCEPT OF THE PROPOSED COVERT CHANNEL

In this thesis, we propose a MAC layer storage covert channel that would ideally work in an environment as illustrated in Figure 6. This figure represents two stations embedded in an 802.11 infrastructure network but at the same time exchanging information through the use of a covert channel. This is the ultimate goal of our research, although it was not fully achieved.

An alternative configuration as described in Chapter IV, in which the covert channel and an Ethernet connection were in use simultaneously, through the use of two network

adapters, was successfully implemented. The reason for implementing a simplified channel is related to the limitations presented by the available hardware. We only had one wireless network card adapter in each station, and for the covert channel to be functional, that card had to be set to Monitor mode. This mode does not allow a simultaneous connection to the infrastructure network. One possible solution would be installing a second wireless network adapter to connect to the infrastructure network.

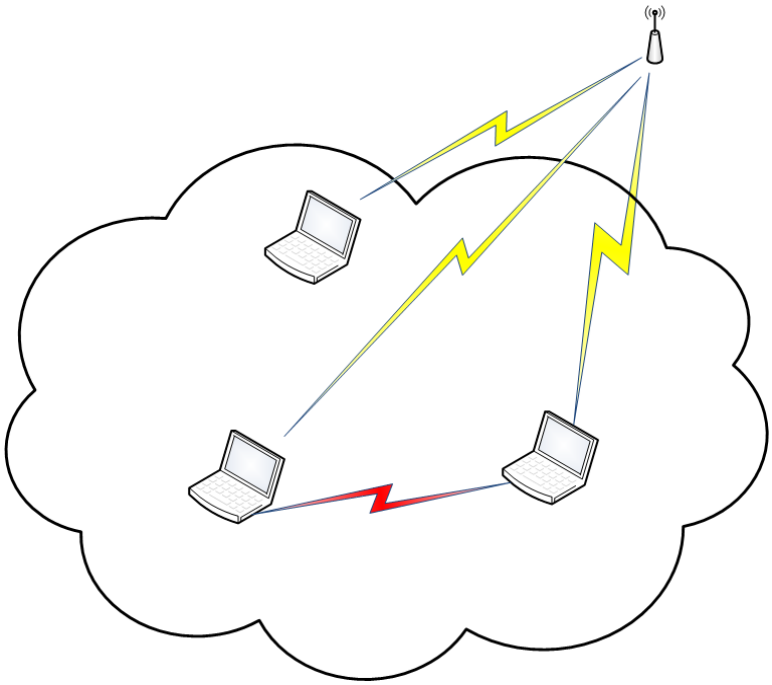


Figure 6. Network topology in which the covert channel operates.

Once the covert channel is established, the robustness of the channel becomes relevant. The error performance of the channel depends on the network traffic and potential collisions and loss of frames. To ensure the proposed

channel has some degree of resilience to transmission errors, the use of forward error correction (FEC) and interleaving was considered and tested. A convolutional code of rate $2/3$ and constraint length of four was employed.

In summary, this chapter provided an overview of the IEEE 802.11-2007 standard, protocol and network architectures, frame construction and most common frame types. A classification of covert channels and existing work on covert channels related to our work were presented. A conceptual description of the proposed covert channel was provided.

III. DESIGNING THE COVERT CHANNEL

A covert channel can be used as a means to convey information without other parties realizing that there is a hidden communication taking place. In this thesis we investigate the implementation of a covert channel in an IEEE 802.11-2007 wireless network.

An overview of different covert channels and the architecture, as well as some frame formats of the IEEE 802.11-2007 standard were covered in Chapter III. A new covert channel that utilizes specific bits in the MAC header of an 802.11 network is presented in this chapter, and the problem of implementing a functional covert channel is addressed.

A. NETWORK MONITORING

In the previous chapter we discussed the various types of frames in 802.11 networks. For the construction of the proposed covert channel, we examine these frames to identify one or more fields in the MAC header that are suitable for information transfer. In order to do so, we must first choose the type of frame suitable for this purpose. The necessary analysis to make a sound decision is provided in the following section.

1. Type of Frame Analysis

A heavily used 802.11 network on campus is monitored to collect frame traffic on multiple channels. The network channels monitored were channel 1 and channel 9. From the MAC frame traffic collected, channel 1 is found to be the

one with most traffic volume and number of users. We collected over 22 million packets to analyze the following frame basic characteristics.

The first characteristic we examined was the type of frame that would best suit our needs. Ideally, we want a frame that is short in length, common in occurrence, and still valid if some bits are changed. Additionally, its presence in bursts should not be a rare event. These features are desirable for achieving a reasonable throughput while providing covertness.

The results of our analysis are shown in Figure 7 as a pie chart, which represents the frequency of occurrence of different types of frames. The data frames are dominant, followed by CTS, ACK and beacons. The "others" refers to the sum of all other frames that represent less than 1% individually. From this plot, we can clearly see that two types of control frames matching our needs stand out, the ACK and the CTS.

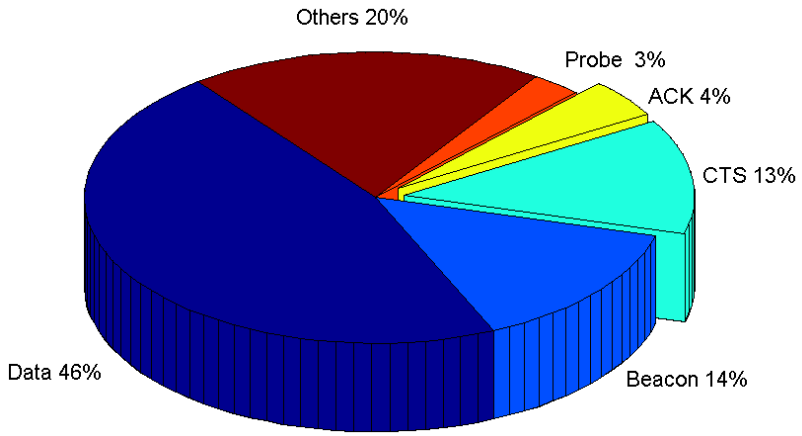


Figure 7. Frequency of occurrence of the monitored frame types

We chose to use CTS for building the proposed covert channel as the CTS traffic volume is large and is of same frame size as ACK. The next desirable characteristic of the frame is the burst length, i.e., the consecutive occurrence of the same frame type in an 802.11 wireless network under normal operating conditions.

2. Sequence of Frames Analysis

Initially, one aspect taken into consideration was the importance of having sequences of ACK or CTS originating from the same station. This became irrelevant since the frame does not contain a source address, and the destination address of the forged frame can be manipulated as necessary. The importance of frame sequence is relevant when we are concerned with the detectability of our channel. One could detect a rogue station by observing the received power level, the signal-to-noise ratio (SNR) of the received frames, and recognizing the fact that the frames originated from the same location [25]. Such analysis might work if the wireless stations are stationary, which defeats the purpose of mobility, but it may be a typical scenario for a limited time, as in an office space or conference room.

In the traffic we collected, long sequences of consecutive frames of the same type, either ACK or CTS, directed to different stations were observed. The sequence length versus the frequency of occurrence of the ACK and CTS, respectively, are illustrated in Figures 8 and 9. We excluded any sequence length with less than two occurrences. For reference, a maximum length of 252

consecutive CTS frames was recorded once, but it was clearly a unique event in all of the monitored traffic.

In Figure 8, we notice the high incidence of short sequences of consecutive frames (up to 10 consecutive frames), and lower occurrence of lengths above 40 consecutive ACK frames.

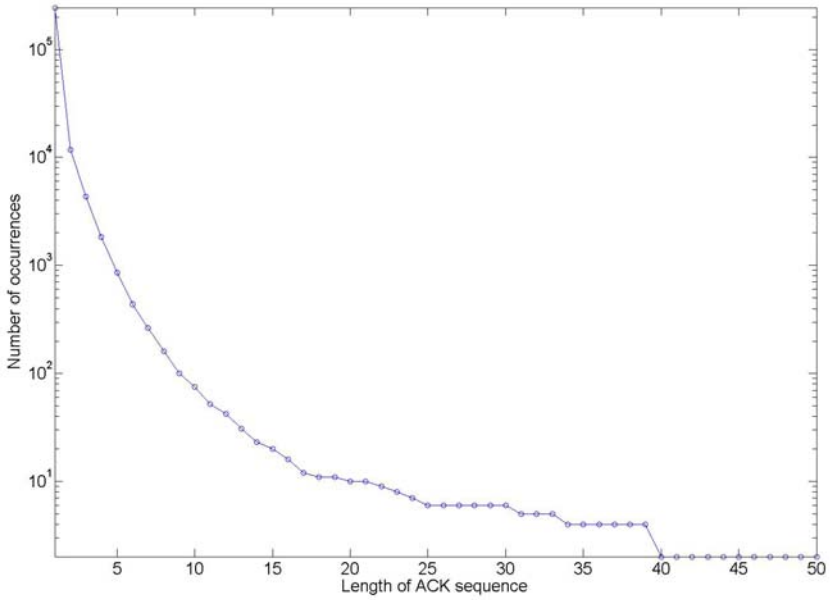


Figure 8. Distribution of acknowledgement (ACK) frame burst length.

From Figure 9, it is clear that CTS is more likely to have long consecutive sequences.

One abnormality noticed during the traffic analysis was the presence of "unexpected" frames among the collected traffic. By unexpected, we mean that some of the captured frames contain a protocol version number other than zero,

which should be the default value [6]. The following section examines the protocol version field.

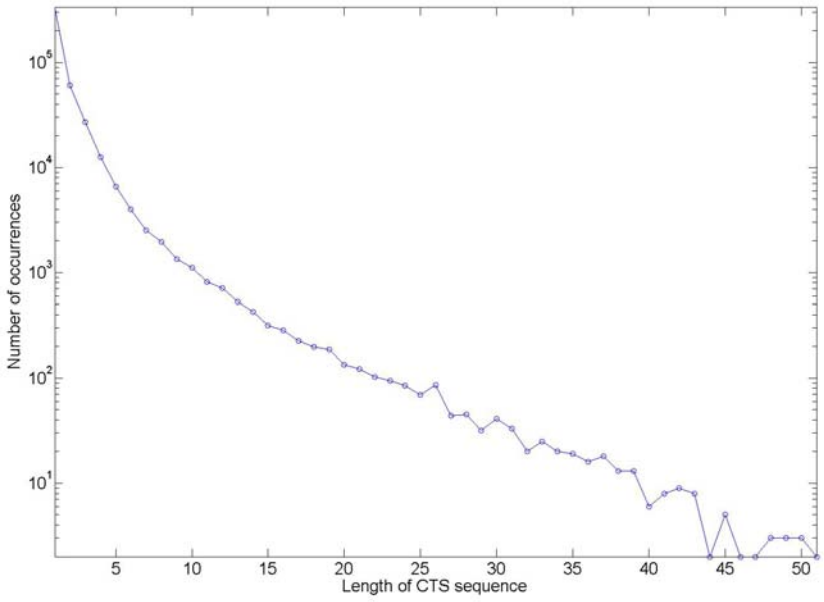


Figure 9. Distribution of clear to send (CTS) frame burst length.

3. Protocol Version Field Analysis

The zero value for the protocol version field is set by the standard, and at the time of this writing it has not been changed. This led us to look into it with more attention, since this is one potential field that can be used by the proposed covert channel.

The results of traffic analysis conducted on the protocol version field are contained in Tables 3 to 5. From the traffic data collected in our experiment, we selected 6,189,701 frames to examine the protocol version field, as shown in Table 3. A vast majority of the frames

(6,182,148 frames or 99.88%) were found to contain version 0. Frames containing version numbers 1, 2, and 3 were very few in number.

Table 3. Protocol version field values of all captured frames.

Version #	Total #	Total %
0	6,182,148	99.88
1	2,880	0.05
2	3,347	0.05
3	1,326	0.02
ALL	**6,189,701**	**100**

We conducted the same analysis on the two specific frames of interest, the ACK and CTS frames, and the results are listed in Tables 4 and 5. As we can see, the incidence of protocol version other than zero is quite low.

Table 4. Protocol version field values of the captured ACK frames.

Version #	Total #	Total %
0	1,269,379	99.95
1	288	0.02
2	296	0.02
3	8	0.00
ALL	**1,269,971**	**100**

Table 5. Protocol version field values of the captured CTS frames.

Version #	Total #	Total %
0	2,997,890	99.99
1	88	0.00
2	310	0.01
3	13	0.00
ALL	**2,998,301**	**100**

Presence of the protocol version other than 0 is puzzling. On the one hand, this means that the security mechanisms in the access point may not be performing a thorough analysis of the frame headers; a properly functioning security mechanism should block the frames with non-zero protocol version field. On the other hand, if we intended to use this field as a means for the covert channel, the existence of other stations transmitting a value other than zero would be a source of noise in the channel.

To insure that frames contain non-zero version field are not a result of malformed, corrupted or fragmented packets, we further examined the frame traffic. We found that frames with a protocol version higher than zero contained mismatched frame check sums; i.e., they were formed due to bit errors. An example of such a frame can be seen in Figure 10, a Wireshark [26] capture, where we highlighted the version field and the failed checksum.

4. Choosing the Frame Type

In the process of choosing a frame for the covert channel, several frames were considered, such as RTS and

ACK. These frames could serve as well as the CTS, but they were found to be less frequent than CTS. Also, among these three frames, RTS is the longest one with 20 bytes, and the CTS and ACK have only 14 bytes. For this reason we narrowed the options to ACK and CTS. The smaller the number of bits we have to transmit to send a covert message, the more efficient the channel becomes.

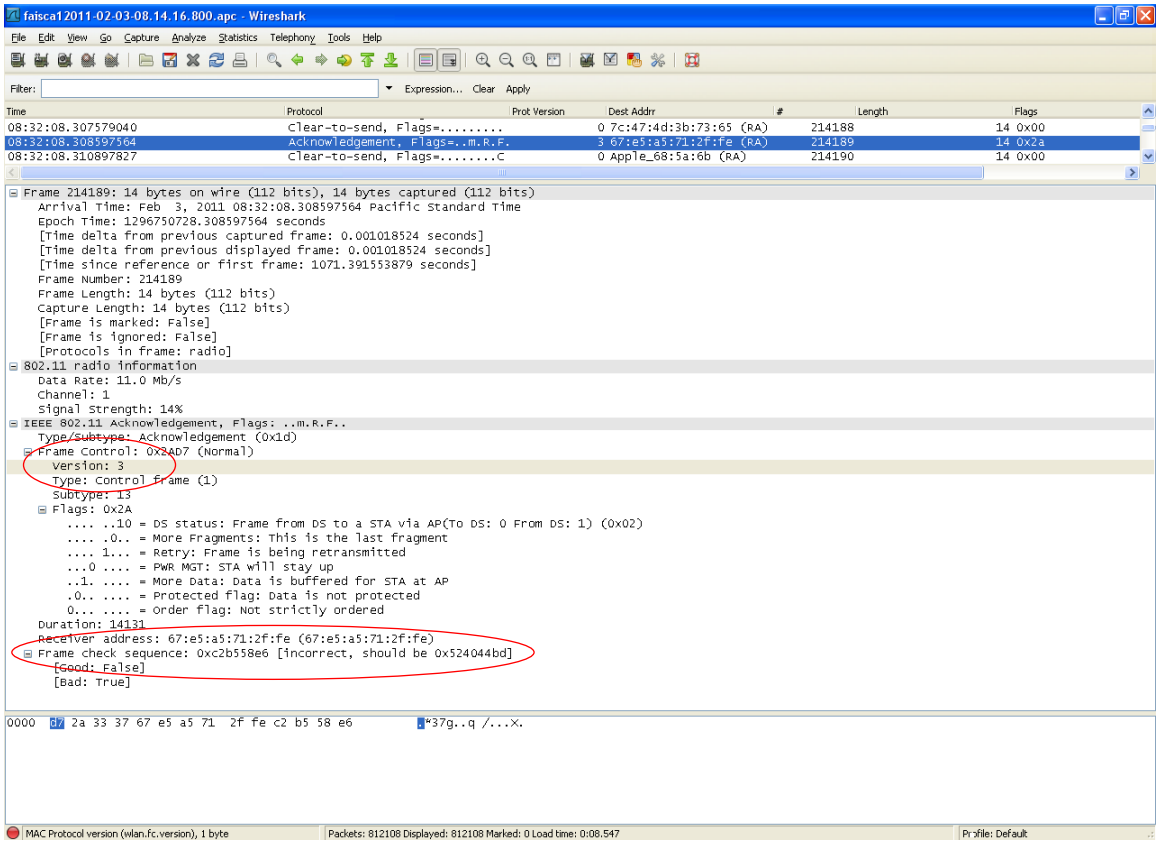


Figure 10. Wireshark capture of a CTS frame with PV = 3 and incorrect check sum.

From monitoring of frame traffic on the campus wireless network and empirical analysis, we found that the CTSs occur with a frequency twice that of the ACKs. The monitoring was conducted in different traffic scenarios,

ranging from low traffic periods to high levels of utilization of the network. By choosing CTS, we can minimize the chance of causing a traffic anomaly based on the type and frequency of packets flowing through the network. Also, we already found that the presence of a long burst of CTS's is not uncommon in 802.11 networks. During frame traffic monitoring, we frequently observed long sequences (up to 50) of consecutive CTS's. Of course, this sequence length would not allow us to send that many bits in a row. A way around this issue is to slow down the rate at which we generate and transmit the forged frames. This would drastically reduce the throughput but would increase the stealthiness of our channel.

Since CTS and ACK have a similar frame structure, it is easier to switch from one to the other, according to our objectives. The main concept of the proposed covert channel applies equally to both frames. It is even possible to have one end of the channel transmitting ACK frames and the other transmitting CTS frames without any loss or degradation of performance. Alternating frame types, such as transmitting a forged ACK followed by a forged CTS is also viable. Many other variations are also feasible.

The fact that both CTS and ACK frames do not contain a source address also contributes to a higher level of stealthiness since it is not possible to immediately identify the source of the transmission.

B. PROPOSED COVERT CHANNEL

1. MAC Header Manipulation

In the proposed covert channel, we use two bits in the protocol version field of the MAC header of an 802.11 CTS packet to carry hidden information. The first two bytes in the MAC header is the frame control field. The generic two-octet long frame control field with the protocol version field highlighted is shown in Figure 11.

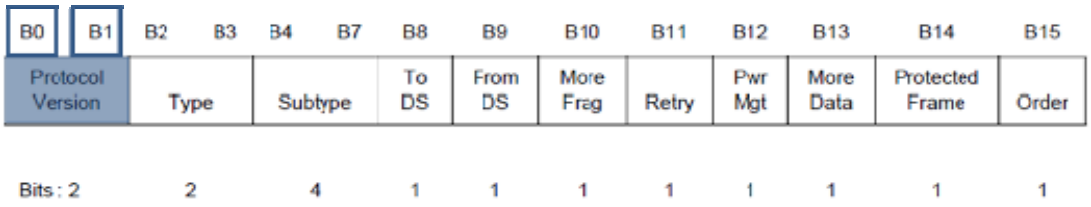


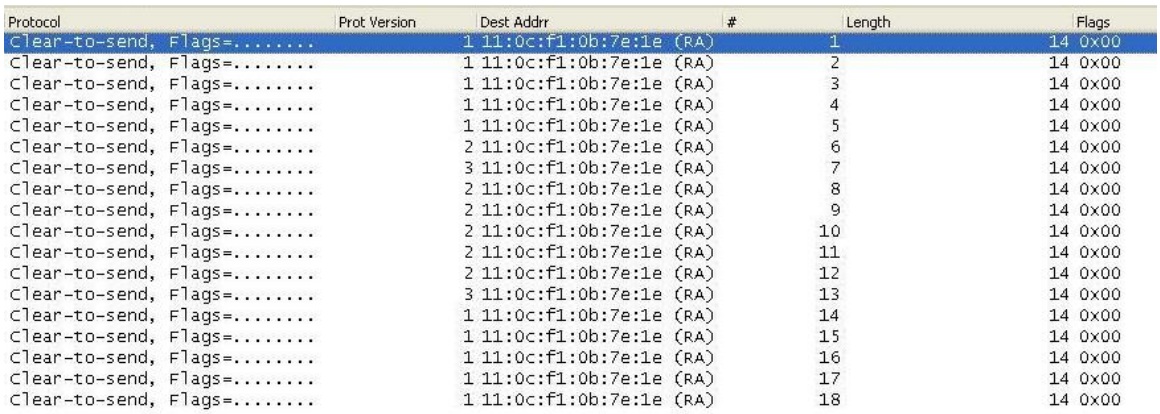
Figure 11. Frame control field [From 6].

The proposed covert channel uses the protocol version bits in a variety of ways to signal the beginning and end of the transmission as well as to carry the information, one bit at a time.

In order to facilitate communication in the proposed covert channel, we divided the transmission into three segments: start message delimiter, message, and end message delimiter. The start and end delimiters are realized by transmitting a sequence of five frames with 01 in the protocol version field. The message bits are transmitted using combinations of 10 as binary "0" and 11 as binary "1" in the protocol version field. The message is organized into 8-bit ASCII characters.

An example of this procedure is shown in Figure 12, where a capture of Wireshark is displayed in which we can see the transmission of the ASCII character "A" converted into the binary string "01000001." A total of 18 frames were transmitted as follows:

- five CTS frames with protocol version one (01) mark the beginning of the transmission;
- eight CTS frames corresponding to the binary representation of the ASCII code of character 'A', with protocol version 2 (10) representing a binary zero and protocol version 3 (11) a binary one; and
- five CTS frames with protocol version one (01) marking the end of the transmission.



Protocol	Prot Version	Dest Addrr	#	Length	Flags
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	1	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	2	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	3	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	4	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	5	14	0x00
Clear-to-send, Flags=.....	2	11:0c:f1:0b:7e:1e (RA)	6	14	0x00
Clear-to-send, Flags=.....	3	11:0c:f1:0b:7e:1e (RA)	7	14	0x00
Clear-to-send, Flags=.....	2	11:0c:f1:0b:7e:1e (RA)	8	14	0x00
Clear-to-send, Flags=.....	2	11:0c:f1:0b:7e:1e (RA)	9	14	0x00
Clear-to-send, Flags=.....	2	11:0c:f1:0b:7e:1e (RA)	10	14	0x00
Clear-to-send, Flags=.....	2	11:0c:f1:0b:7e:1e (RA)	11	14	0x00
Clear-to-send, Flags=.....	2	11:0c:f1:0b:7e:1e (RA)	12	14	0x00
Clear-to-send, Flags=.....	3	11:0c:f1:0b:7e:1e (RA)	13	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	14	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	15	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	16	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	17	14	0x00
Clear-to-send, Flags=.....	1	11:0c:f1:0b:7e:1e (RA)	18	14	0x00

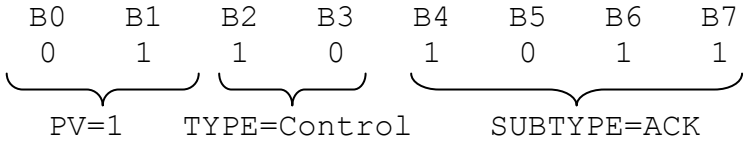
Figure 12. Wireshark capture of an "A" being transmitted using the proposed covert channel.

2. Important MAC Header Parameters

It is important to keep the forged frame as a valid frame to minimize the chance of detection and reduce the likelihood of elimination or blocking by access points,

firewalls or IDSs. Any such device could be looking into the contents of packet headers and discarding invalid ones. The forged CTS generated in our covert channel can be modified to include a valid destination MAC address that exists in the network in which we are operating and a valid checksum. The only deviation from a system generated frame is the PV field value. This can be seen in Figures 13 and 14, where the first highlighted field of the frames, **d6** and **c7**, respectively, represents the type, subtype and protocol version. By inspection we can see that a protocol version 2 is present in Figure 13, indicating the transmission of a binary zero, in our covert channel, and a version 3 is present in Figure 14, signaling a binary one.

The **d6** and **c7** values are the hexadecimal representation of the bits that comprise the first octet in the MAC header. From Table 2, **d6** in Figure 13 is composed as follows:



By taking B7 as the most significant bit, we get 11010110(bin)=d6(hex). The same process applies to the construction of **c7**.

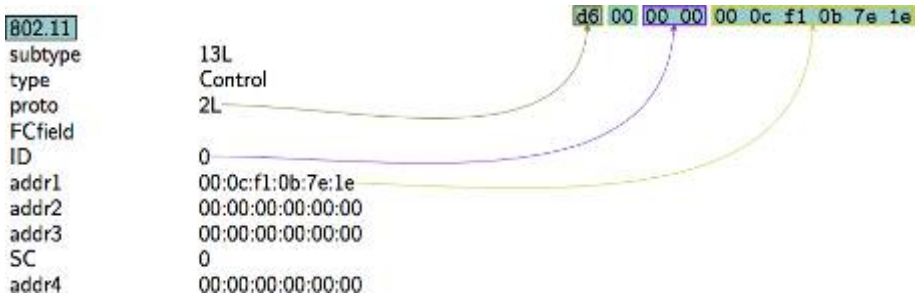


Figure 13. Forged ACK structure.

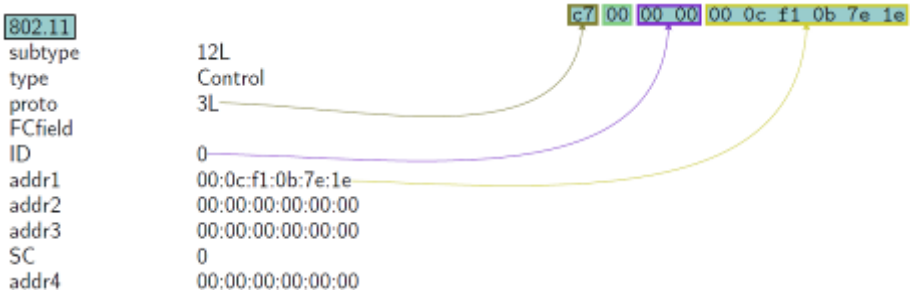


Figure 14. Forged CTS structure.

The ID field or duration field is set to zero in Figures 13 and 14. If the forged frame is an ACK sent by a non-QoS station and has the More Fragments flag set to zero, then this field is also zero. Otherwise, it has a non-zero value [6]. On the other hand, if the forged packet is a CTS, this field indicates how long the referred station in the destination address field has air time to transmit its data, while the remaining stations hold their transmissions during the same period. If this field is set to zero, there are no practical implications to the network. However, if this field contains a non-zero value, all the other stations will hold their transmissions for that amount of time. This could be the basis for a Denial of Service (DoS) attack [27].

C. DETECTING AND DISABLING THE USE OF THE PROPOSED COVERT CHANNEL

Once we are aware of the existence of a covert channel, it is relatively easy to protect against its unwanted use.

In order to limit the use of the proposed covert channel or any of its derivatives that are built upon the same concept, we just have to monitor the PV field in the MAC header. If the PV field is different from zero (00), the packet is discarded. This blocking technique works regardless of the type of packet we forge, since all types of packets have the PV in common [6]. Notice that this blocking rule would only limit (not eliminate) the use of the channel. For example, the frame would be blocked by an AP, but any station in the range of the transmitting station would still “hear” this frame.

In our experiments, since we recorded frames with PV values other than 00 but with an invalid CRC, we cannot infer whether or not the APs are filtering such frames. What we can conclude is that invalid frames with altered PV values exist in the network and that stations within the covert channel's range still “hear” such frames.

Another aspect that could raise suspicion is the presence of a long sequence of frames of the same type to the same MAC address in a short period of time. Although we described a way to circumvent this effect, this is still something to consider and is worth analyzing.

Yet another aspect that could trigger an alarm would be the anomalous increase of the network's traffic during our use of the covert channel. We will examine such a

scenario in Chapter IV, but spacing the transmission of the forged frames in time mitigates this effect. This, however, comes at a cost since the throughput decreases and the transmission period increases.

In this chapter, we presented the results of traffic monitoring in an IEEE 802.11 wireless network and proposed a covert channel. The process of choosing the right frame to forge based on empirical results was explained. The CTS frame was chosen, and some considerations about the frame choice, its strengths and weaknesses were made. The basic premise of the proposed covert channel is to use the PV field in the MAC header for message transmission. Aspects related to the detectability and mitigation were also discussed. The test bed model used for the experiments and a description of the developed code and the analysis of experimental results is presented in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENTS AND RESULTS

In this chapter, we analyze three parameters of our channel: stealthiness, error robustness, and throughput. The intent is to present the results of experiments conducted using a proof-of-concept covert channel program developed by the author.

A. TEST BED

For conducting tests, we used two laptops with the same hardware configuration, using a PCM 3COM 3CRPAG175 with an Atheros chip AR5212 as the wireless network adapter. One laptop was used as transmitter (Station A) and the other one as passive monitor (Station B). Station A was running Backtrack4 as the operating system (OS) as well as some additional software described in Appendix A. Station B ran Windows XP SP2 and the monitoring program used was Airopeek NX, version 3.0.1 [28].

In Chapter II, we described the ultimate goal of this thesis as having two stations that are part of an infrastructure wireless network, communicating between them through a covert channel. Although the concept is fairly simple, the practical implementation is not. For that reason, and also due to time constraints, our approach for the practical tests consisted of having two stations located in the physical area of an infrastructure 802.11 network but not connected to it, trading messages between them using the proposed covert channel. The setup used for the experiments is illustrated in Figure 15.

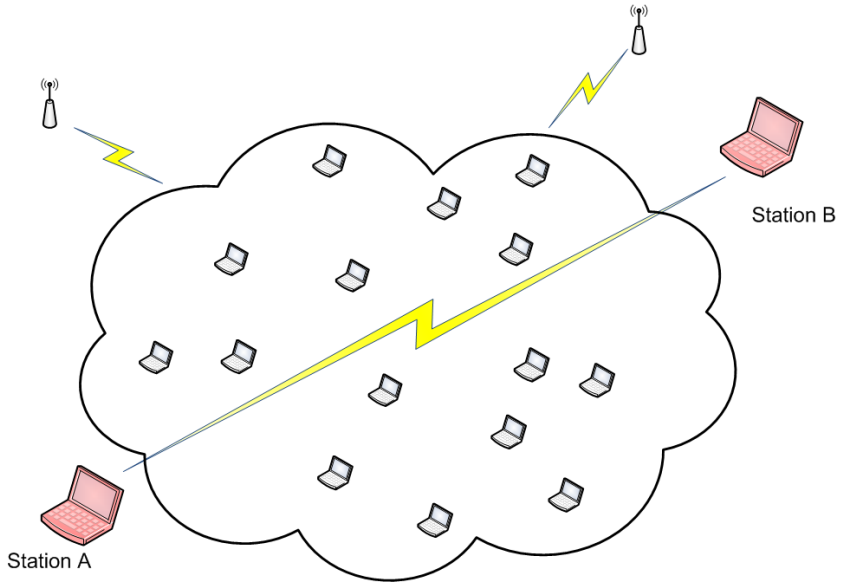


Figure 15. Network topology used in the experiments.

B. CODE DESCRIPTION

In order to implement the proposed covert channel, we developed the necessary code to forge, transmit, and receive frames. To provide error robustness, we used convolutional coding along with interleaving.

Python [13] was the chosen programming language due to its simplicity, available libraries and extension modules that facilitated our task. For the OS, a Linux environment was elected as being more flexible, open source and GNU licensed. The chosen distribution was Backtrack4. Diverse documentation on this OS flavor can be found online. All the additional software (see Appendix A) is also under GNU licensing, so no proprietary software was used to implement the covert channel.

In order to simplify the use of our covert channel, a graphical user interface was used. Since this is a proof-

of-concept effort, implementation of a half-duplex chat room environment seemed reasonable to meet our objectives. This way, we made use of an open-source chat environment previously developed by Wolfman and Filth [29]. The visual interface is used almost unchanged; several internal routines and processes were extensively altered. Effort was put into making the program applicable to realistic application scenarios. A small description of the code follows. In our code we wrapped the covert channel in a friendly GUI, so it looks and operates as a basic chat console.

The code is divided into three major processes running simultaneously in a virtual sense, meaning the processor alternates between all processes in a very small amount of time. This is crucial to the code performance; the code optimization was on our mind but did not take a high priority. Figure 16 is a simple representation of the major blocks constituting the final code. A main program is initiated, along with the loading of several libraries and definition of variables. One of the most important libraries is Scapy [30], a Python packet manipulation program that enabled us to listen and disassemble frames as well as forge our own frames. The GUI is built using Tkinter. A screen capture of the GUI is shown in Figure 17. The various menus and the welcome message are visible, as well as an example of a transmitted and a received message. The transmitted message is the first line, identified with user John, and the second line is the received message, identified with Eve.

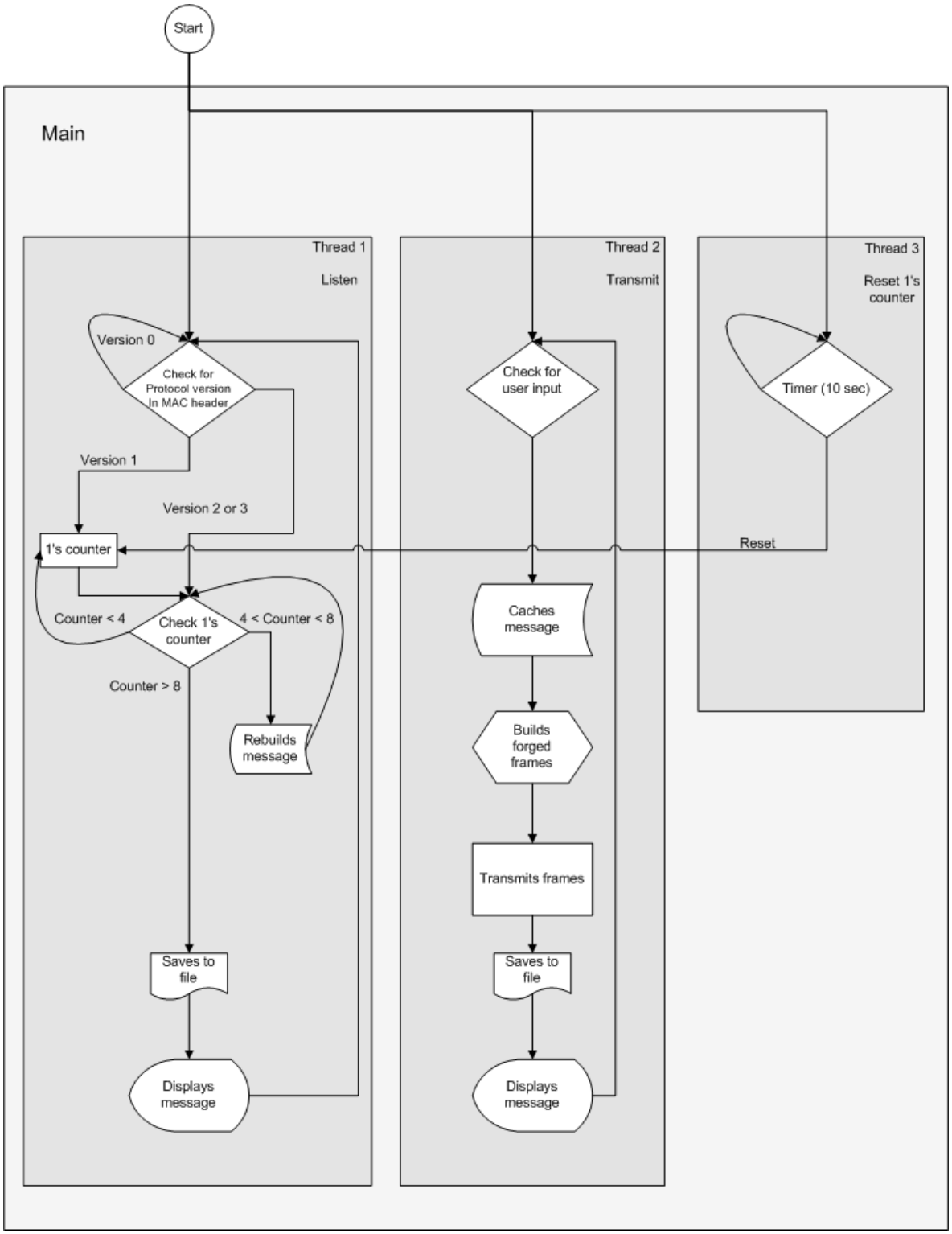


Figure 16. Flow chart of the covert channel code implementation.

Following the flow direction on Figure 16, we then move to Thread1, where we initiate the listening part of our program. Here we filter the frames of interest, identify the beginning and end of the covert communication, and write the resulting message to a log file after converting the recovered string of bits to ASCII characters. The routine responsible for converting the bits to characters and storing them to the log file is:

```
def conv(bin2):
   bl=[bin2[i:i+8] for i in range(0, len(bin2), 8)]
   final=''
   for z in range(0,len(bl)):
        final2=chr(int(bl[z],2))
        final=final+final2
   timestamp='('+"%.19s" % str(datetime.now())+') '
        user='Received message'+': '
        txt = timestamp+user+final+'\n'
        f = open(file.name, 'arb',5)
        f.write(txt)
   f.close()
```

Thread2 corresponds to the transmitting part of the code. We continuously scan the log file, where all the keyboard inputs are saved, check for an update in the file; and if one is detected, we build our binary string, forge the frames, and transmit them. This way, all the received and transmitted messages are saved in the log file with a time stamp and identification of message originator.

During the transmission period of time, we set an internal control flag to 1 in order to suspend the listening routine, thus avoid listening to our own transmission.

Finally, Thread3 handles possible discrepancies in the identification of the beginning and end of the covert communication. The other version 1 frames (with bad checksums) circulating in the network become noise to our version 1 frames forming the start and end delimiters. Thread3 is responsible for filtering out these unwanted frames. The Python code segment of Thread3 is:

```
def treset():
  while True:
       global magic
       c=magic
       tm.sleep(10)
       if c-magic==0 and magic<5:
            magic=0
```

A complete listing of the code is provided in Appendix B.

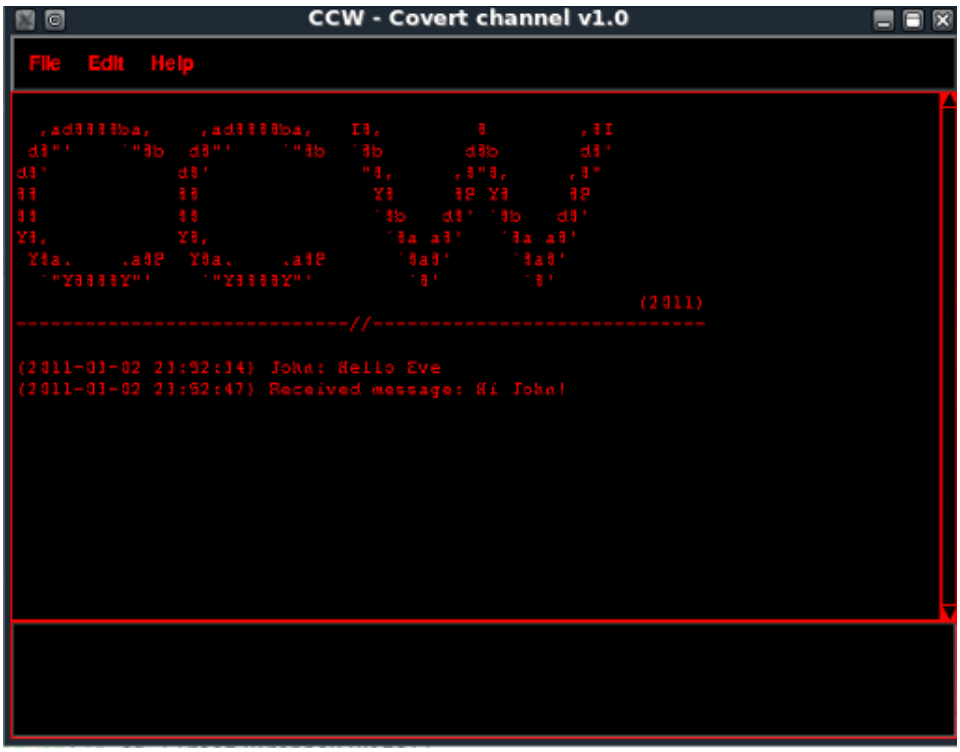


Figure 17. Covert channel GUI screen capture.

C. RESULTS

Frame traffic was recorded over operational wireless networks during week days in order to capture the real-world scenarios.

Three different scenarios were considered and tested. All scenarios consisted of transmitting similar messages during approximately the same time of day. The difference between the scenarios is the way the data was transmitted, since we varied the type of frames used and applied different error mitigation mechanisms.

It is important to notice that Stations A and B were operating in the ad-hoc mode of operation, i.e., outside the infrastructure wireless network being monitored. As a result, the mechanisms in the 802.11 standard designed to minimize collisions are not entirely observed. CSMA/CA was still used since it is a built-in functionality of the wireless adapter. The stations transmit without any coordination from the access point. This likely causes collisions, and thus frame losses, which are interpreted as errors for analysis purposes.

There were two types of messages used during the tests. The first message was a classic steganographic sentence used during WWII by a German spy [31]:

> Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects for pretext embargo on by-products, ejecting suets and vegetable oils.

The sentence has a total of 1408 bytes. The second type of message sent is a 25,000-bit long sequence of binary 1s.

These messages were transmitted under three scenarios. In the first scenario, the messages were sent without any error control.

A demonstration of the sentence transmission is presented now using the chat room GUI. The result of a correct reception of a message sent over the covert channel is shown in Example 1, and a totally unreadable message is shown in Example 2. Both messages were sent over a Scenario 1 environment in channel 1 with no error correction.

Example 1:

(2011-03-04 01:43:52) Received message: Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects for pretext embargo on by-produce, ejecting suets and vegetable oil.

Example 2:

(2011-03-04 01:48:56) Received message: ,_yeóªt,—®;__ ƒ'ÜU›_Õ‡M_-µ[B4□?oÒÍa gdB□óïkC@»4□Ai□__CÄ_eÒ\¥)_?7R€_x,ÚM¿ F□•)-*ep>¹_KXQÕT\…Y{6ŒW7DõøH1ì€tÇžc‹D‡ÉQ_

Forward error correction (FEC) is used to improve robustness of transmitted messages in the second scenario. The last scenario consisted of forward error correction and bit interleaving to further enhance robustness.

1. Error Performance of the Covert Channel

CTS frames are used to carry the message in the PV field. This yields a throughput of approximately one bit per transmitted frame. Actually, it is slightly less than one bit per frame, since we have an overhead of ten frames to mark the beginning and end of the transmission. Channels 1 and 9 of the network were monitored for frame traffic.

Before conducting the analysis, it is important to define what we considered to be an error. In this thesis,

an error is the loss of a payload (information) bit. If we send one payload bit per frame, then the loss of one frame corresponds to one error. The reception of malformed frames, with invalid checksum indicating bit flips, is classified as a lost frame in our analysis.

a. Channel 1

In Figure 18(a) we can see the profile of the traffic collected for a period of about ten hours on channel 1. The percentage of errors detected upon reception of the test sentence is displayed in Figure 18(b). Finally, the percentages of errors recorded for the 25,000 long bit sequence of binary 1s is illustrated in Figure 18(c). The plots are time aligned. The width of the bars in Figures 18(b) and 18(c) indicate the time it takes to transmit the complete sequence.

Summarizing this analysis, we observed an average error of approximately 3% for the sentence and 2% for the sequence of ones over a total of 30 sets of transmissions.

One desirable characteristic we want to preserve in a covert channel is the stealthiness of message transmission; that is, we try to hide as much as possible such that the use of the covert channel remains unnoticed.

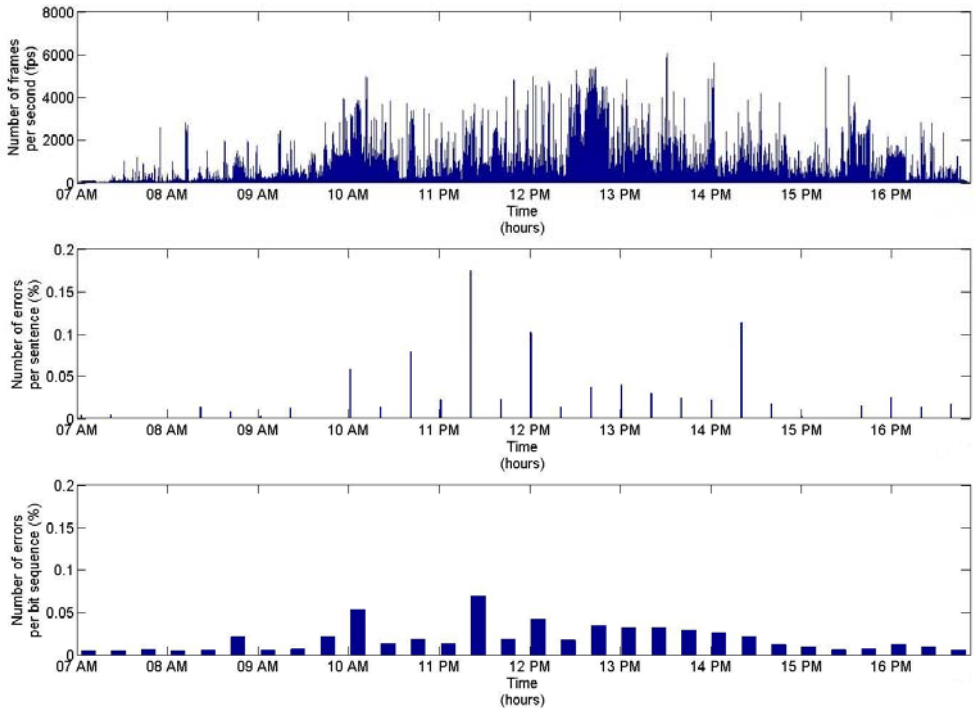


Figure 18. Network traffic profile and percentage of errors for sentence and sequence receptions in channel 1.

Figure 19 is a partial magnification of the traffic profile shown in Figure 18(a), where the black (lower) line represents the normal network traffic, and the red (upper) line shows the normal traffic plus the traffic due to covert (forged) frames. As we can see in Figure 19, the difference between the red line and the black line corresponds to the amount of traffic added by the use of the covert channel. Since the network traffic is fairly heavy in channel 1, the presence of the covert channel is not obvious; our traffic just blends in with the overall traffic.

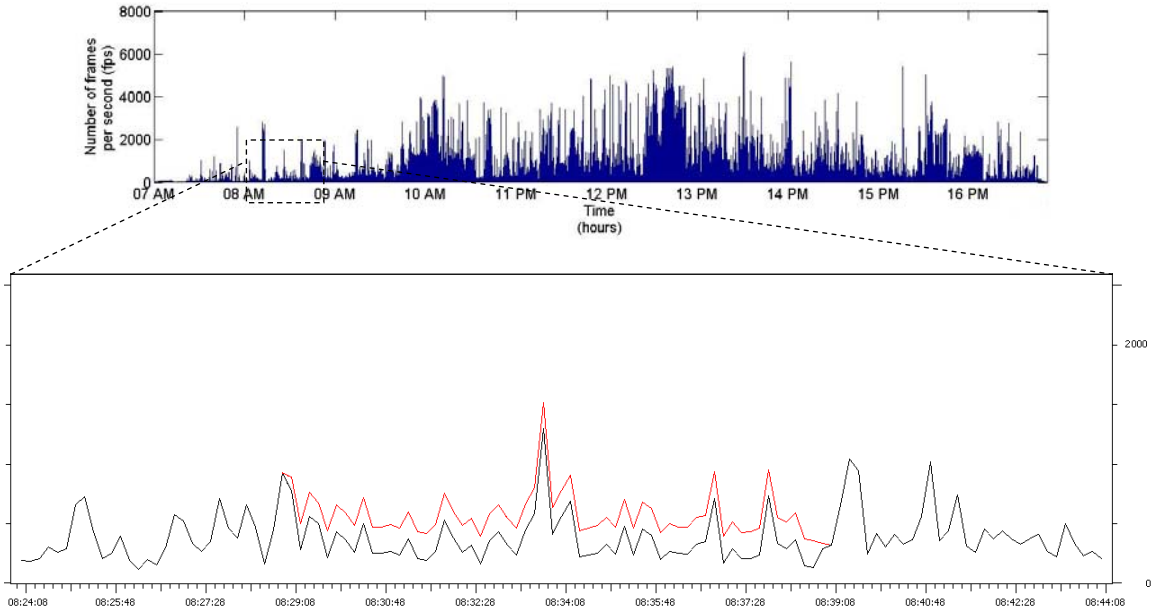


Figure 19. A selected portion of network traffic profile for channel 1.

b. Channel 9

We repeated the same experiment using channel 9 instead of channel 1. Here, we did not expect any heavy traffic; thus, no significant information is gained regarding traffic profile shaped by users. For that reason we reduced the sequence of ones from 25,000 to 2,500 bits in order to have a large number of sequences in a shorter amount of time. The results in Figure 20 correspond to a two-hour period of monitoring channel 9 without any covert activity.

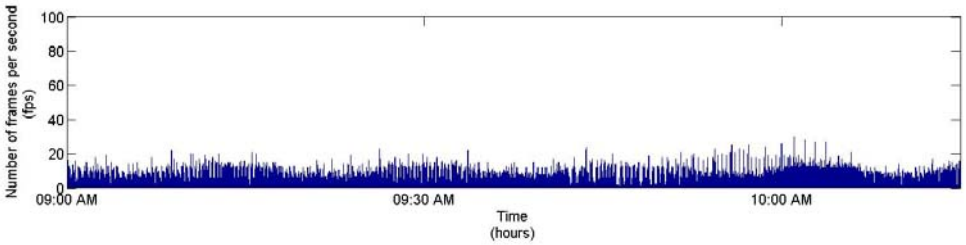


Figure 20. Network traffic profile of channel 9.

The results of the traffic profile with covert channel activity are shown in Figures 21 and 22. Figure 21(a) is the graphical representation of the number of frames per second in circulation in the network between 9AM and 7PM on a weekday. The percentage of errors in the sentence reception is shown in Figure 21(b). In Figure 21(c) we have the representation of the percentage of errors for the 2,500 bit long sequence.

A zoomed in view of the normal traffic in the network versus the covert channel traffic is displayed in Figure 22. In this case the difference is large, and the presence of the channel is easily revealed. The red line represents the traffic due to the covert channel, whereas the black line is the normal traffic in the network. This situation is exactly what we do not wish in a real-world application.

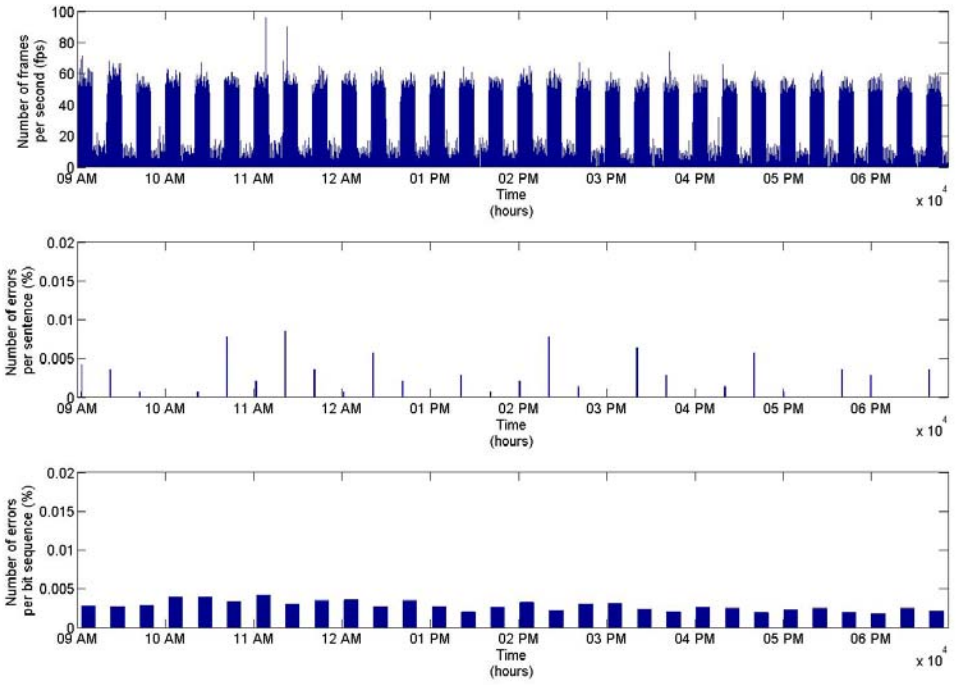


Figure 21. Network traffic profile and percentage of errors for sentence and sequence receptions in channel 9.

The stealthiness of the channel can be improved by spacing the transmission of forged frames. How the covert traffic can be made less visible by introducing spacing between frames is illustrated in Figure 23. This of course reduces the throughput. Segment (a) in Figure 23 corresponds to normal frame transmission with no additional spacing between the frames. For this segment the total transmission time was approximately two minutes at an average of 30 frames per second (fps). In segment (b) frames are sent once every two seconds, resulting in a total transmission time of 2 hours and 12 minutes. Finally,

segment (c) is shown only partially; we sent one frame every four seconds for a total transmission time of 4.5 hours.

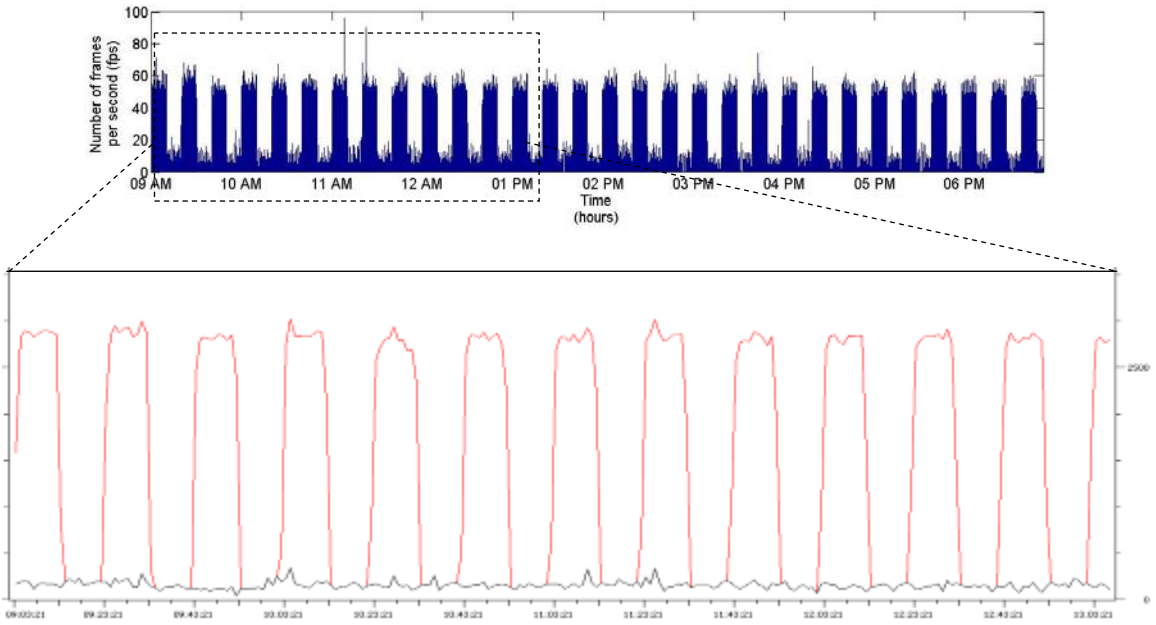


Figure 22. A selected portion of network traffic profile for channel 9.

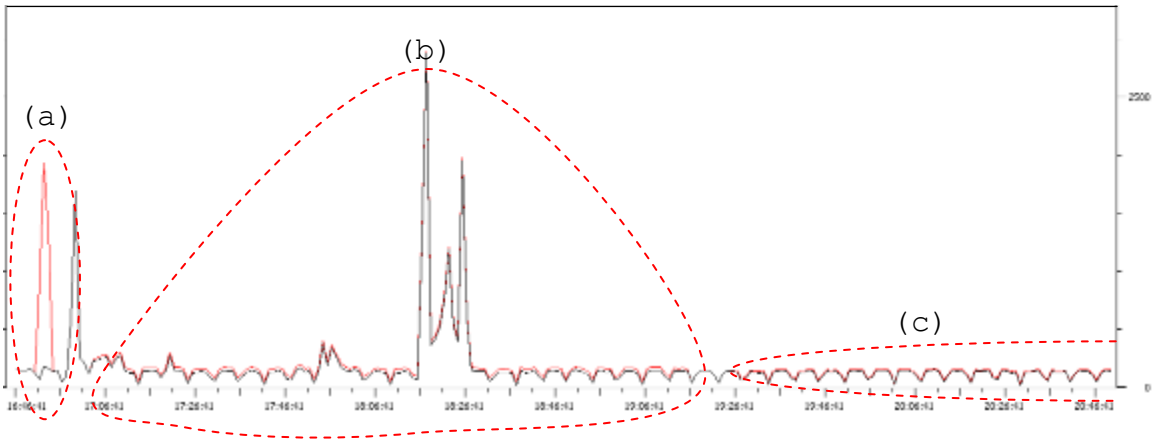


Figure 23. Zoom of network traffic profile for channel 9 using three different delays in forged frame transmission.

The important aspect is that the difference between the legitimate and covert traffic becomes smaller and smaller as the spacing increases; at some point, it is possible to make it almost invisible as we extend the spacing. On the other hand, the throughput is degrading proportionately.

Another technique to camouflage our use of the covert channel is to space the forged frames transmission in a non-uniform way instead of sending the frames at regular time intervals. Although considered, this variation was not tested.

2. Error Performance of the Covert Channel With Forward Error Correction

We now introduce forward error correction in order to reduce the number of errors in the covert channel.

There are several options for implementing FEC: block codes such as Hamming and Reed-Solomon, convolutional codes, turbo codes, or low density parity check codes. In this thesis, a convolutional code was used for error correction.

A convolutional encoder takes an m-bit message and encodes it into an n-bit symbol. The ratio m/n is the code rate. In our case a code rate of $2/3$ was used, meaning the encoded message will be one and a half times as long as the original message. This increases the time needed to transmit the same message as before since a larger number of channel bits is being sent.

Another important parameter in convolutional coding is the constraint length. This parameter, k, represents the

number of bits in the encoder memory that affect the generation of the n output bits [32]. A constraint length of four is used for our experiments.

In order to deal with the presence of burst errors in the channel, in association with the convolutional coder we also used bit interleaving [33,34]. This consisted of breaking the coded message in blocks of eight bits and building a matrix with each block in a different row. By reading the matrix out by column, from top to bottom, we generate a new string of bits, effectively interleaving all the eight bit blocks. The number of rows depends on the length of the message we are transmitting. This process is shown in Figure 24.

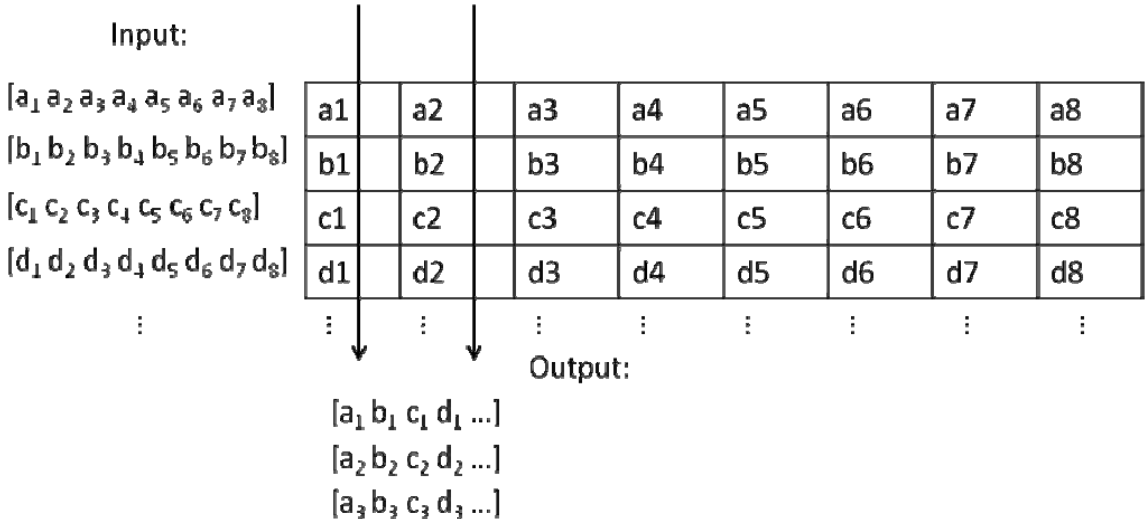


Figure 24. Bit interleaving process.

Forward error correction is typically applied to a transmission of a stream of bits sent and received sequentially. In our case, however, the bits are embedded into independent frames, which are prone to loss. As a

result, when a frame is lost, the receiver has no indication that a bit was missing. Consequently, we now need to know exactly which frames were lost in order to apply the FEC correctly. Different approaches were tested, and the results are reported below.

a. Alternating CTS and ACK

A rudimentary mechanism for determining the location of the lost frames can be implemented by alternating the frame type, accomplished by sending alternate ACKs and CTSs. Essentially, we are using the subtype field in the MAC header to accomplish this; the PV field is still the carrier of the covert information.

This approach effectively emulates a 1-bit sequence number. As soon as we lose more than one frame in a row, the entire sequence is corrupted, and the error correction scheme is unable to correct the errors (lost frames). A better scheme is needed.

b. Alternating CTS and ACK Using Sequence Numbers

We propose to use the eight flag bits in the frame control field of the MAC header to obtain a longer sequence number, which makes determining the location of lost frames an easier task. However, it is important to state that applying this use of the flag bits will increase the probability of detection of the covert channel since unexpected flag attributions will be present. This was not further investigated, but we are aware of the increased risk of detection taken when pursuing an increase in the channel’s error performance. In order not to use the flag

bits, one could use the type and subtype fields of the MAC header. As shown in Table 2, the IEEE802.11 standard defines some bit combinations of the subtype field as "Reserved." Exploring these combinations could be an option; although, we did not test it.

Figure 25 is a representation of how we accommodated the information and sequence bits within the MAC header. The blue squares represent our covert channel bits. These bits are used in the same way described in Chapter III: the first bit (B0) signals the presence of the channel and the second is payload (B1). The red circles refer to the sequence bits, which are placed in the flag bits of the frame control field. Given that we have eight flags, this gives us a total of 256 possible sequence numbers. This alone provides a reasonable amount of protection against a long burst of frame losses when compared to the previous approach.

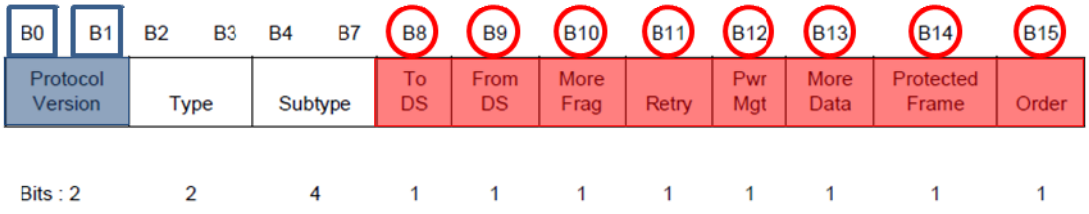


Figure 25. Representation of the frame structure using the flag bits for sequencing.

Figure 26 is an illustration of Wireshark capture of part of the transmission sequence of the sentence. Looking at the flag field, we can see how the hexadecimal values are increasing sequentially.

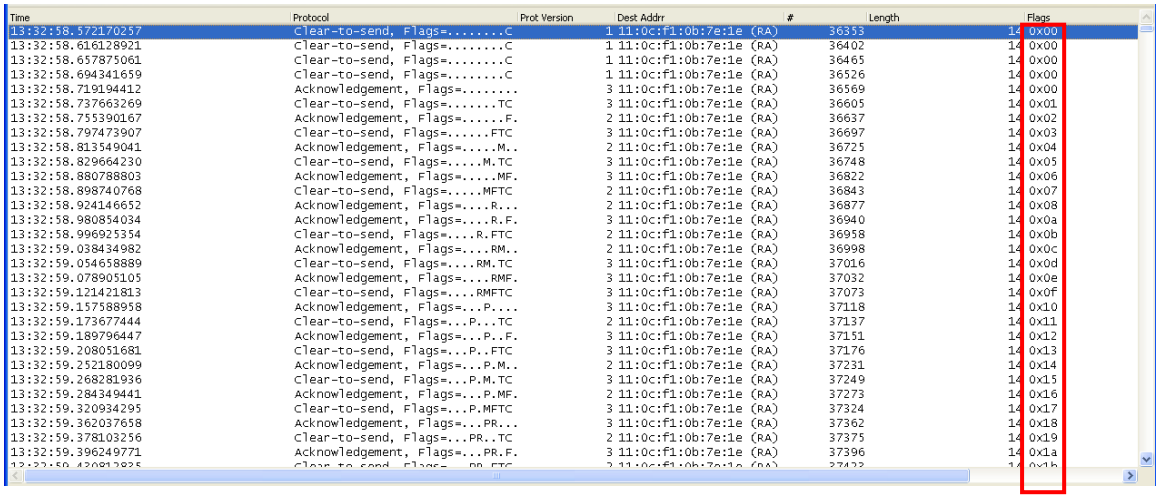


Figure 26. Wireshark capture of transmitted forged ACK and CTS frames using flag bits for sequencing.

This way of using the flag bits does not affect the traffic profile in the network since the number of forged frames is still the same.

The percentage of errors as a function of 15 repeated transmissions of the sentence in channel 1 over a period of four hours is shown in Figure 27. The length of the transmitted sentence is now 2,112 bits long because we applied a $\frac{2}{3}$ rate encoder on a 1,408-bit string. The red stems (x) represent the number of errors detected in the received sentence, and the blue stems (o) the number of errors in the received sentence with FEC. In most cases the number of errors drops to zero or is significantly reduced.

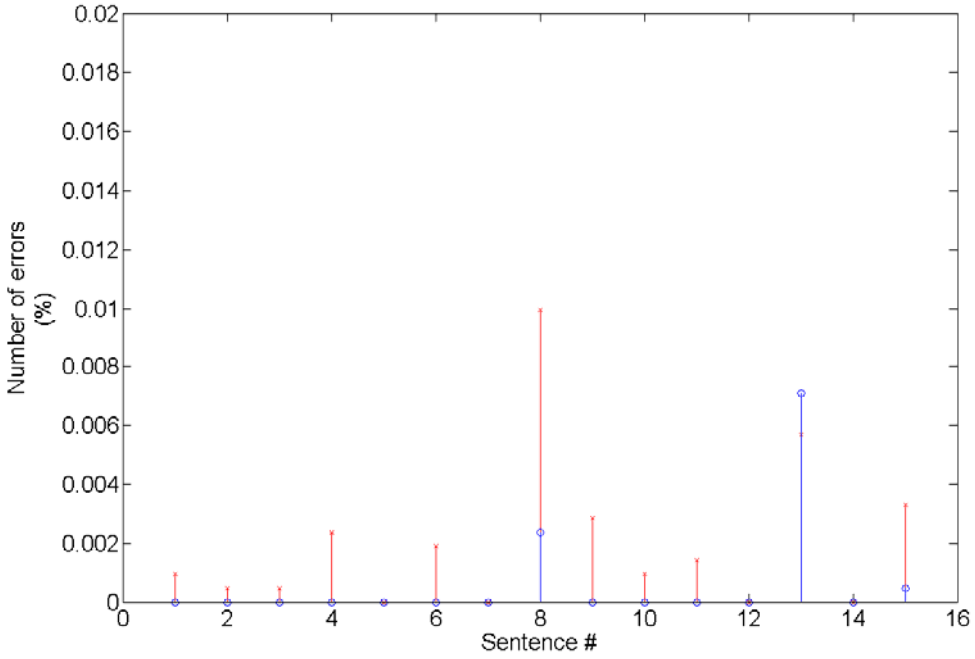


Figure 27. Percentage of errors before (red) and after FEC (blue) per received sentence, using flag bits for sequencing.

This is consistent with our expectations. We have one outlier in that for the 13th repetition of the sentence we got a larger number of errors with FEC.

We recorded a total of 67 errors in this experiment (without FEC), which translates into an average of 4.5 errors per sentence, or an average error percentage of 0.21%. After the execution of FEC, the total number of errors dropped to 21, resulting in an average of 1.4 errors per sentence, or an overall average of 0.09% relative to the 1,408 bits of the original message. This was an improvement of more than two-fold. However, this gain was the direct result of having to transmit more bits to send the same message when compared to the first scenario with no FEC, thus reducing the data rate.

The next test used the sequence of ones. The original length of 25,000 bits becomes 37,500 bits long after encoding. The error values for the 15 repetitions of the bit sequence are presented in Figure 28. Notice that the scale on the y-axis is different from that in Figure 27 since larger values were plotted. The two largest values, corresponding to sequence numbers 3 and 4, are most likely the result of losing synchronization during the decoding of the bit string, leading to an uncontrolled increase in the number of errors. Recall that if the correct frame sequence is lost, the rest of the binary string is corrupted. An example of such an event is the loss of the marker that indicates the beginning and end of the channel use, the PV 1 values.

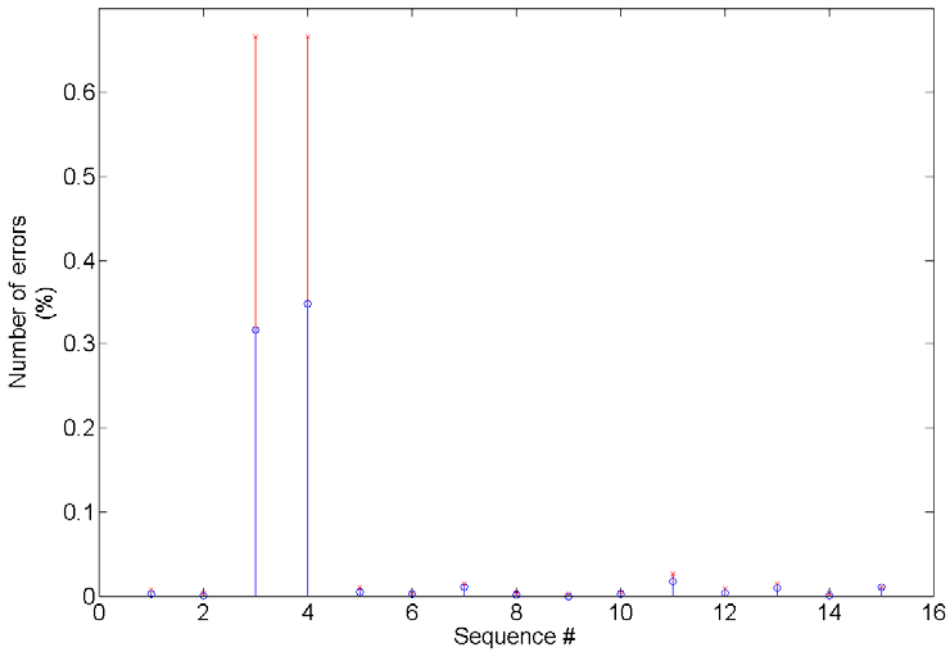


Figure 28. Percentage of errors before (red) and after FEC (blue) per received sequence, using flag bits for sequencing.

Excluding the outliers, the total amount of errors at the receiver for the remaining 13 sequences was 4,211. This gives us an average of approximately 324 errors per repetition, or 0.86% of the total 37,500 bits transmitted per sequence. After the FEC, the total number of errors dropped to 2,502, or 0.77% relative to the original 25,000 bit long sequence.

3. Error Performance of the Covert Channel With Forward Error Correction and Interleaving

In this scenario, we continued to use alternating sequences of CTS and ACK frames as well as FEC. We now consider sending more than one bit of information per forged frame. The proposed structure is illustrated in Figure 29. The blue squares indicate payload bits, and the red circles are sequence numbers. The green diamond (B0) indicates the presence of the covert channel. Bits B1, B8 and B9 form the sequence number yielding a sequence length of 8. Bits B10-B15 form the payload of six bits to carry the message.

Figure 29. Representation of the frame structure using three bits for sequencing and six bits for payload.

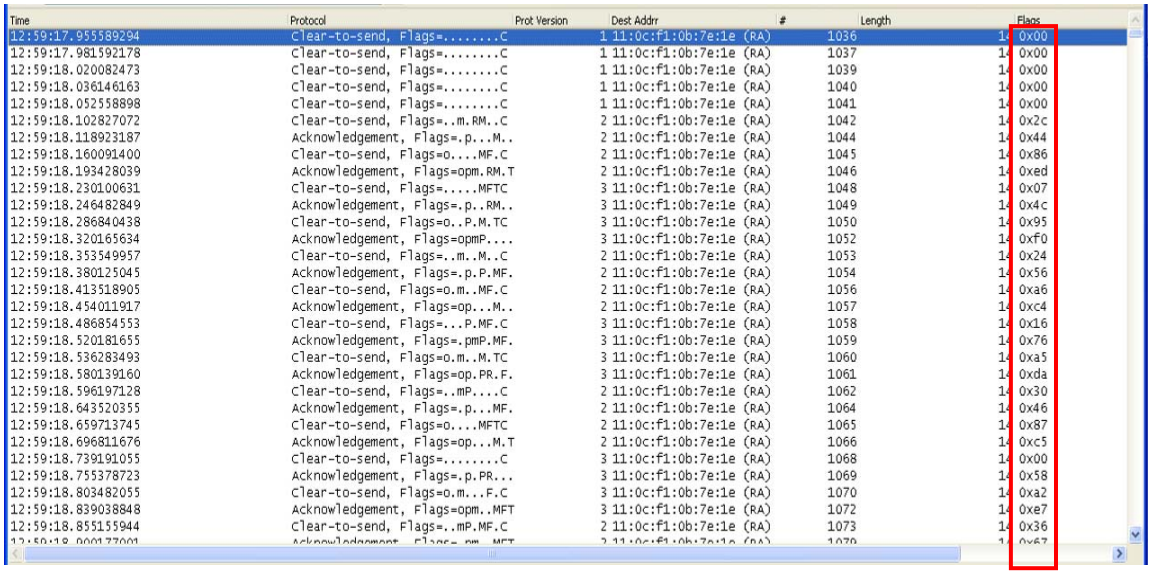
Since each frame now carries six information bits, the loss of one or more frames has a bigger impact on the number of errors in the channel. In order to mitigate this effect, we interleave the bit string resulting from the

convolutional coder. Figure 30 is a schematic representation of this idea. At the output of the convolutional coder, we interleave the bits in groups of eight bits, as shown in Figure 24. This results in a new string of zeros and ones which goes into the covert channel processing block. Here the string is separated in groups of six bits, and each group becomes the payload of the forged frames.

Figure 30. FEC and interleaving block diagram.

Notice that only information bits are encoded and interleaved; in this implementation the convolutional coder is applied after we have the complete message we want to transmit. In other words, first we capture the entire message, then we encode it, interleave it, and finally run the resulting string through the covert channel. The frame is forged as follows: six information bits are placed in the selected flag bits, three other bits are used for sequence numbers, and the first PV bit is set to one, indicating the use of the covert channel.

Figure 31 is a display of a Wireshark capture of some transmitted frames. Notice how the flag values of successive frames change in a non-sequential way since every forged frame has different contents for these fields.



Time	Protocol	Prot Version	Dest Addr	#	Length	Flags
12:59:17.955589294	Clear-to-send, Flags=.....C	1	11:0c:f1:0b:7e:1e (RA)		1036	0x00
12:59:17.981592178	Clear-to-send, Flags=.....C	1	11:0c:f1:0b:7e:1e (RA)		1037	0x00
12:59:18.020082473	Clear-to-send, Flags=.....C	1	11:0c:f1:0b:7e:1e (RA)		1039	0x00
12:59:18.036146163	Clear-to-send, Flags=.....C	1	11:0c:f1:0b:7e:1e (RA)		1040	0x00
12:59:18.052558898	Clear-to-send, Flags=.....C	1	11:0c:f1:0b:7e:1e (RA)		1041	0x00
12:59:18.102827072	Clear-to-send, Flags=..m.RM..C	2	11:0c:f1:0b:7e:1e (RA)		1042	0x2c
12:59:18.118923187	Acknowledgement, Flags=.p...M..	2	11:0c:f1:0b:7e:1e (RA)		1044	0x44
12:59:18.160091400	Clear-to-send, Flags=o....MF.C	2	11:0c:f1:0b:7e:1e (RA)		1045	0x86
12:59:18.193428039	Acknowledgement, Flags=opm.RM.T	2	11:0c:f1:0b:7e:1e (RA)		1046	0xed
12:59:18.230100631	Clear-to-send, Flags=.....MFTC	3	11:0c:f1:0b:7e:1e (RA)		1048	0x07
12:59:18.246482849	Acknowledgement, Flags=.p..RM..	3	11:0c:f1:0b:7e:1e (RA)		1049	0x4c
12:59:18.286840438	Clear-to-send, Flags=o..P.M.TC	3	11:0c:f1:0b:7e:1e (RA)		1050	0x95
12:59:18.320165634	Acknowledgement, Flags=opmP....	3	11:0c:f1:0b:7e:1e (RA)		1052	0xf0
12:59:18.353549957	Clear-to-send, Flags=..m..M..C	2	11:0c:f1:0b:7e:1e (RA)		1053	0x24
12:59:18.380125045	Acknowledgement, Flags=.p.P.MF.	2	11:0c:f1:0b:7e:1e (RA)		1054	0x56
12:59:18.413518905	Clear-to-send, Flags=o.m..MF.C	2	11:0c:f1:0b:7e:1e (RA)		1056	0xa6
12:59:18.454011917	Acknowledgement, Flags=op...M..	2	11:0c:f1:0b:7e:1e (RA)		1057	0xc4
12:59:18.486854553	Clear-to-send, Flags=...P.MF.C	3	11:0c:f1:0b:7e:1e (RA)		1058	0x16
12:59:18.520181655	Acknowledgement, Flags=.pmP.MF.	3	11:0c:f1:0b:7e:1e (RA)		1059	0x76
12:59:18.536283493	Clear-to-send, Flags=o.m..M.TC	3	11:0c:f1:0b:7e:1e (RA)		1060	0xa5
12:59:18.580139160	Acknowledgement, Flags=op.PR.F.	3	11:0c:f1:0b:7e:1e (RA)		1061	0xda
12:59:18.596197128	Clear-to-send, Flags=..mP....C	2	11:0c:f1:0b:7e:1e (RA)		1062	0x30
12:59:18.643520355	Acknowledgement, Flags=.p...MF.	2	11:0c:f1:0b:7e:1e (RA)		1064	0x46
12:59:18.659713745	Clear-to-send, Flags=o....MFTC	2	11:0c:f1:0b:7e:1e (RA)		1065	0x87
12:59:18.696811676	Acknowledgement, Flags=op...M.T	2	11:0c:f1:0b:7e:1e (RA)		1066	0xc5
12:59:18.739191055	Clear-to-send, Flags=.....C	3	11:0c:f1:0b:7e:1e (RA)		1068	0x00
12:59:18.755378723	Acknowledgement, Flags=.p.PR...	3	11:0c:f1:0b:7e:1e (RA)		1069	0x58
12:59:18.803482055	Clear-to-send, Flags=o.m...F.C	3	11:0c:f1:0b:7e:1e (RA)		1070	0xa2
12:59:18.839038848	Acknowledgement, Flags=opm..MFT	3	11:0c:f1:0b:7e:1e (RA)		1072	0xe7
12:59:18.855155944	Clear-to-send, Flags=..mP.MF.C	2	11:0c:f1:0b:7e:1e (RA)		1073	0x36

Figure 31. Wireshark capture of transmitted forged ACK and CTS frames using three bits for sequencing and six bits for payload.

a. FEC Without Interleaving

We first test the channel using FEC and no interleaving. The error values for the 15 repetitions of the sentence are depicted in Figure 32. The average number of errors per repetition was 85, or 0.27%, of the total amount of bits sent per sentence. After tracking the sequence numbers and correcting the bit sequence, the final number of errors was reduced to 35 or 0.16%.

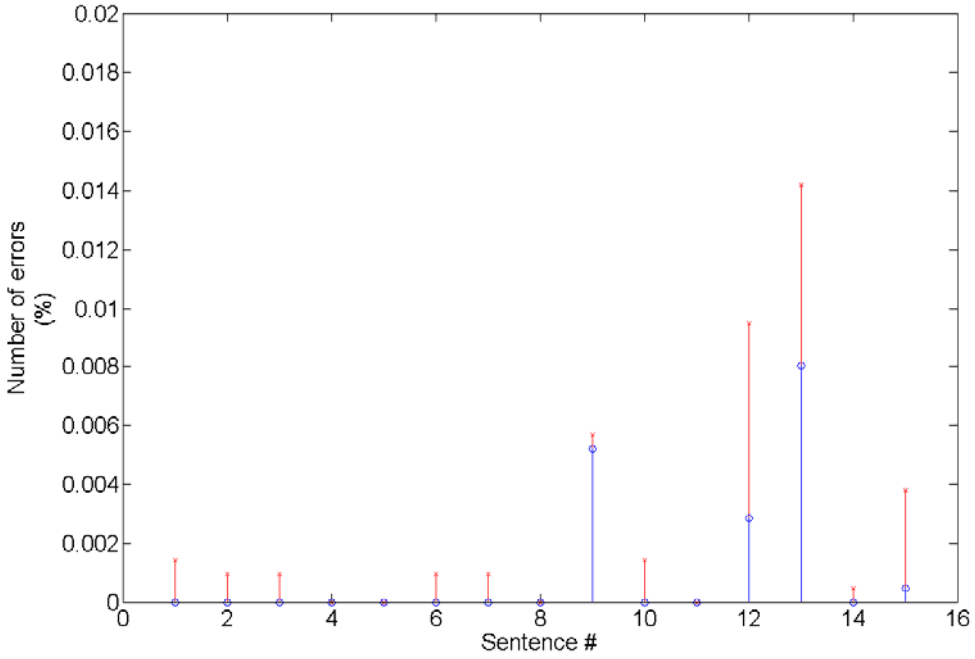


Figure 32. Percentage of errors before (red) and after FEC (blue) per received sentence without interleaving.

b. FEC With Interleaving

In this experiment, we use three bits for sequencing. For a payload of six bits, the loss of one packet has a bigger effect in the error performance of the channel. For this reason we resorted to the use of interleaving.

The percentage of errors per sentence repetition can be seen in Figure 33. From this figure we notice an outlier at repetition 12, actually gaining errors after the FEC. This was an isolated event and was excluded from this analysis. The result is an average number of 1.53 errors per repetition, or 0.07%, of the total amount of bits sent per sentence. Following the sequence number tracking, de-

interleaving and correcting the bit sequence, we see that the total number of errors is reduced to zero. These are significant results; however, the sample space is small, and we cannot conclude that this level of robustness will be achieved in every reception.

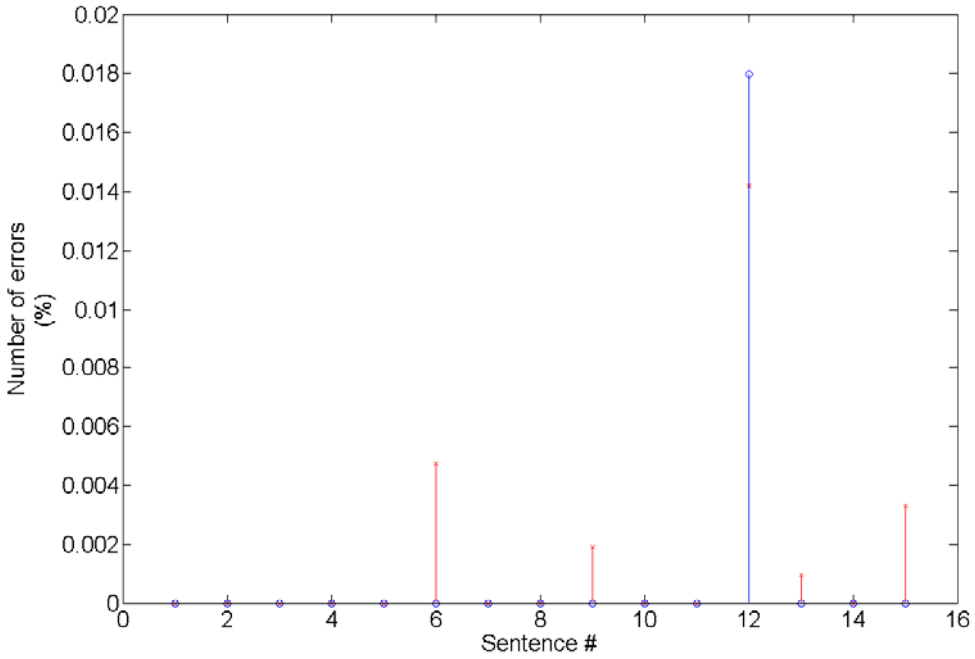


Figure 33. Percentage of errors before (red) and after FEC (blue) per received sentence with interleaving.

D. THROUGHPUT ANALYSIS

In order to evaluate the throughput offered in each scenario, the rate at which the frames were transmitted was measured. This was done using Airopeek [28] and by averaging the rate of the forged frames on a per second (fps) basis. The measured transmission rates may have large variations and may reach zero in some cases because sometimes no frame is sent during an entire second. Depending on the network usage at the time, the frame rate

varies significantly. Another factor responsible for this variation is the continuous adjustment of the maximum data rate of the network as dictated by the channel conditions. For IEEE 802.11b networks, the maximum network data rate possible values are 1, 2, 5.5, and 11 Mbps [6].

To obtain a benchmark for performance comparison, we first determine the maximum data rate possible for the covert channel under optimal conditions. The conditions we assume are:

1. The channel is ideal with no errors;
2. There is only one station with frames to transmit;
3. We use a data rate of 2 Mbps, the highest possible for 802.11b control frames (basic rate set) [6].

The medium access scheme has to obey some predetermined timing constraints set by the standard. Figure 34 is a graphical representation of the timing requirements for transmitting a frame.

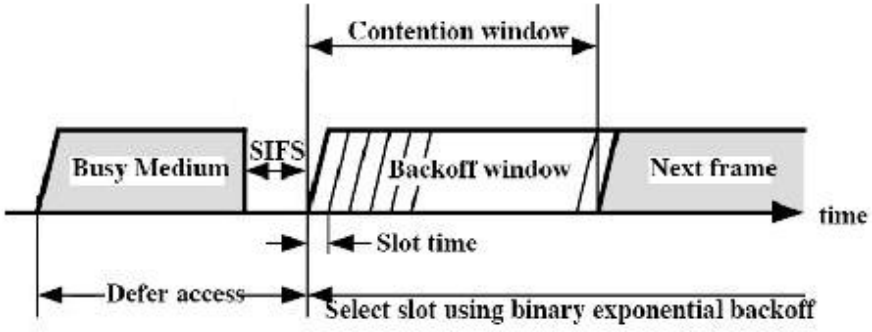


Figure 34. Timing constraints in an 802.11 frame transmission [After 35]

Applying the work of Xiao and Rosdhal [36] and Jun et al. [37] to the proposed covert channel, the minimum amount of time necessary to transmit a forged CTS $t_{\min}$ can be expressed as

$$t_{\min} = t_{SIFS} + \frac{N_{cwin} t_{slot}}{2} + t_{CTS} \quad (1)$$

where t_{SIFS} is the short interframe time, $t_{CTS} = \frac{14 \times 8}{11 \times 10^6} = 10.18$ µs is the transmission time of the 14-byte CTS frame, N_{cwin} is the maximum size of the contention window, and t_{slot} is the slot time.

From the standard we use $t_{SIFS} = 10$ µs, $N_{cwin} = 31$ and $t_{slot} = 20$ µs. This yields $t_{\min} = 376$ µs, corresponding to a maximum of 2659 forged frames per second. At one bit per frame, the maximum bit rate is 2659 bps; at six bits per frame, we get 15.954 kbps. The measured throughput values, however, are significantly smaller.

Having established a benchmark, we now determine the experimental throughput results. In the first experiment, we were able to transmit one bit of information in each forged frame, but we have the overhead of the start and end delimiters, a total of ten signaling frames. The measured average frame rate was 61 frames per second. Since each frame represents a bit, and considering our message payload of 1408 bits, we transmit a total of 1418 bits. At 61 fps this corresponds to a total transmission time of 23.25 sec and a useful bit rate or throughput of 60.5 bits per second (bps).

The next experiment introduced FEC, and although we did not change the frame construction, the measured average

frame rate is smaller. This is due to the additional processing introduced with the sequencing of the frames. Instead of the previous 61 fps, we now have 43 fps being transmitted. Also, the total number of bits needed to send the message increases to 2122 due to the use of the convolutional coder. This corresponds to a total transmission time of 49.4 seconds, yielding a useful throughput of 21.3 bps.

In the last experiment, we transmitted six bits per forged frame and introduced the use of interleaving. The measured average transmission rate was 32 fps, and transmitting the same 2122 bits as before, we obtain a total transmission time of 11 seconds. The resulting throughput value is 127.4 bps. The measured results are summarized in Table 6.

Table 6. Measured throughput values compared to the channel data rate

	Useful bit rate (bps)
Max. bit rate (Theoretical)	2659
Without FEC and 1 bit payload	60.5
With FEC and I bit payload	21.3
With FEC and 6 bit payload	127.4

In summary, results of testing the proposed covert channel were presented in this chapter. The Python code

used to implement the covert channel was briefly described. Results of experiments were presented, with emphasis on robustness to errors, channel covertness and achieved throughput. A summary of the conclusions made in previous chapters, significant results and recommendations for future work are given in the next chapter.

V. CONCLUSIONS

The IEEE 802.11-2007 standard was the subject of our work in this thesis. This standard was first introduced in 1997, yet it is still expanding and is one of the most widely used wireless networking standards. According to an industry report, in 2012 over a billion devices will be shipped with technology based on this standard onboard and the number is projected to be over two billion in 2014 [38]. For this reason, we think it is important to evaluate the possible weaknesses and vulnerabilities in order to determine relevant security challenges. The particular focus in this work is covert channels, which have the characteristic of being hard to detect unless we know in advance what we are looking for. Consequently, continuous research and investigation in this field are essential.

A previously undocumented 802.11 covert channel was implemented and tested in this thesis. We introduced and discussed the basic concept of the proposed covert channel. We then implemented the channel in a Linux environment and tested it under different scenarios in order to analyze its robustness, covertness and throughput. The necessary code to implement the channel and a GUI were developed in Python. The results of the experiments were presented and discussed. Considerable effort and resources were put into the development of the code, collection of the network frame traffic, and analysis of a large quantity of recorded network traffic.

A. SIGNIFICANT RESULTS

A new covert channel in wireless networks based on the 802.11 standard was identified. We used the protocol version field in the MAC header to hide and transfer the covert information.

The proposed covert channel was implemented by developing the necessary code in Python. A GUI chat console is used for message transmission. The test bed used for experiments operated in a Linux environment.

Robustness to errors in the covert channel was improved by the use of forward error correction and bit interleaving. Preliminary results indicate significant improvement in the error performance of the channel.

The achieved throughput of the covert channel was measured under three scenarios. The maximum channel data rate is also determined. The case of a 6-bit payload along with convolutional coding and interleaving yielded the highest measured throughput.

B. FUTURE WORK

There are several aspects in which this work can be complemented and improved. One aspect to be potentially explored is to test whether the covert channel can be detected by APs. This would expand the characterization of the channel, providing a better understanding of its range of features.

On the structural side of the proposed channel, other types and subtypes of frames can be investigated and

implemented. Not only the ones directly assigned by the standard, but also the set of types and subtypes classified as reserved (see Table 2).

A further study of error performance would also be beneficial. One can conduct a larger range of tests under different scenarios, including experimenting in a controlled environment where the noise level and frame collision rates can be monitored. The possibility of using other error correction mechanisms is also of interest. Possible techniques include the use of a repetition code, where the frames are sent a fixed number of times. Knowing the number of repetitions and comparing the expected number to the actual number of received frames would allow us to find the location of lost frames. A spreading code would work in a similar way, but occasionally the bit values are inverted.

In this work, we only used one wireless network adapter. Future research should consider the possibility of having two wireless network adapters installed in each station in order to explore the scenario described in Chapter II, where the stations are part of an infrastructure network while simultaneously using the covert channel.

A future effort should consider the extension of the concept behind the proposed covert channel to wireless networks based on other standards, such as IEEE 802.16 (WiMAX)[39] or Long Term Evolution (LTE)[40].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A

The necessary steps taken to execute the developed software and implement the proposed covert channel are described in this Appendix.

We started by installing a fresh copy of the Backtrack4 Linux distribution on the stations. Once the installation was complete, we updated it by executing the following commands:

- *apt-get update*
- *apt-get upgrade && apt-get autoclean*

This action downloaded and installed all the available updates and cleaned the unnecessary installation files.

Now we disable the wireless interface ath1 in order to reinstall drivers that are compatible with our needs for raw packet injection. To do that, we execute:

- *ifconfig ath1 down*
- *svn –r 4073 checkout http://svn.madwifi-project.org/madwifi/trunk/ madwifi-ng*
- *cd madwifi-ng*
- *wget http://patches.aircrack-ng.org/madwifi-ng-r4073.patch*
- *patch –N –p 1 –I madwifi-ng-r4073.patch*
- *./scripts/madwifi –unload*
- *make*
- *make install*

This loads and sets the madwifi drivers as the interface drivers.

We now need to blacklist the old drivers, so only the new ones are loaded. This is accomplished by:

- go to /etc/modprobe.d/blacklist

- edit the file and add this line at the end:
  blacklist ath5k
- reboot

After rebooting the machine, we logon as root, start the GUI by executing:

- *startx*

and initialize the wireless adapter in the following way:

- *modprobe ath_pci*
- *airmon-ng stop ath0*
- *airmon-ng start wifi0*
- *aireplay-ng -9 ath0* (to test the injection capability)

Now the machine is almost ready to run our code, but some additional software is still needed. The Python version used in this thesis was 2.5.2 [13]. Also, Psycho 1.6 [41] was installed in order to speed up the execution of the Python code. We also installed Scapy version 2.1.0 [27] to enable frame forging. To have the ability to generate PDF files of the captured packets using Scapy (e.g., Figures 13 and 14), we execute the following commands:

- *apt-get install tcpdump graphviz imagemagick python-gnuplot python-crypto python-pyx*

To install PDF Reader 9.4 for linux:

linux-english-version 9.4.tar.bz2
(http://get.adbe.com/reader/otherversions)

APPENDIX B

```
#############################################################################
#This program implements a covert channel by forging and transmitting      #
#control frames in 802.11 networks. It was written using the previous work #
#of Wolfman and Filth for the graphical interface and reception routine.   #
#The forging and transmission routines are original work of Ricardo        #
#Goncalves, as part of the requirements for MSCEE degree at NPS, CA, USA.  #
#March2011                                                                 #
#############################################################################

from __future__ import with_statement # MUST remain at the beginning of the file
import time as tm # used to avoid 100% cpu usage
from Tkinter import * # for the main portion of the program
import threading # to allow simultaneous reading/writing
from datetime import datetime, date, time # for timestamping save file and program output
from tkFileDialog   import askopenfilename, asksaveasfilename # open/save dialogs
import sys # to ensure proper termination
from socket import * #enables the use of sockets
from scapy.all import sniff,Dot11 #loads scapy and the necessary tools to sniff packets
import pylorcon
from fcntl import ioctl
import binascii
import psyco

#set the inject and monitor modes for the wireless adapter (ath0)
wifi=pylorcon.Lorcon("ath0,""madwifing")
wifi.setfunctionalmode("INJECT");
wifi.setmode("MONITOR");

global sending
sending=0
# set the default name for the archive file, which holds the conversations.
class file:
    name='covert.txt'

# now we try to open it, and create it with some default text if it dosent exist
try:
    f = open(file.name, 'r+b')
except IOError:
    f = open(file.name, 'wb')
    f.write('''
  ,ad8888ba,    ,ad8888ba,   I8,        8        ,8I
 d8"'    `"8b  d8"'    `"8b  `8b       d8b       d8'
d8'           d8'             "8,     ,8"8,     ,8"
88            88               Y8     8P Y8     8P
88            88               `8b   d8' `8b   d8'
Y8,           Y8,               `8a a8'   `8a a8'
 Y8a.    .a8P  Y8a.    .a8P      `8a8'     `8a8'
  `"Y8888Y"'    `"Y8888Y"'        `8'       `8'
                                                    (2011)
------------------------------//------------------------------
''')
    f.close()
finally:
    f.close()

#############################################################################
# open the user file...or create it
try:
    f = open('.user', 'r+b')
except IOError:
    f = open('.user', 'wb')
    f.write('Tony') # set your own default here if the file doesnt exist...
    f.close()
finally:
```

```
    f.close()
# define that we want the user in the program to come from the file
class who:
    u=open('.user').read()

#########################################################################
# begin menubar functions
def openf():
    choice=askopenfilename()
    file.name=choice
    #print 'open dialog returned '+choice
    # this can be uncommented for testing proper execution of file dialog

def save():
    n='CCW covert channel v1 '+"%.16s" % str(datetime.now())+'.txt'
    save=asksaveasfilename(initialfile=n) # this is quite straitforward...
    with open(file.name) as f:
        g=open(save, 'wb')
        for line in f.readlines():
            g.write(line) # it copies the file line by line
    #print 'saved file at '+save
    # much like the other dialog, this can be uncommented for testing

def exit():
    sys.exit(0)

# you can change the display name as you wish...it will be saved
def name():
    root=Tk()
    root.config(bg='black')
    l=Label(root, text='Please enter your name below')
    l.config(bg='black', fg='red', bd=0)
    l.pack(side=TOP)
    name=Entry(root)
    name.config(width=14,    bg='black',    fg='red',    insertbackground='red',    bd=0,
highlightbackground='red')
    name.insert(INSERT, str(who.u).strip('\n'))
    name.pack(side=BOTTOM)
    def save(s):
        who.u=name.get().strip('\n')
        with open('.user', 'wb') as f:
            f.write(who.u)
        root.destroy()
    name.bind('<Return>', save)

# help menus are always useful...this one will be too, when i get around to making it
def help():
    win=Tk()
    win.config(bg='black')
    win.title('About')
    say=Text(win)
    say.insert(0.0, '''
  ,ad8888ba,    ,ad8888ba,   I8,        8        ,8I
 d8"'    `"8b  d8"'    `"8b  `8b       d8b       d8'
d8'           d8'             "8,     ,8"8,     ,8"
88            88               Y8     8P Y8     8P
88            88               `8b   d8' `8b   d8'
Y8,           Y8,               `8a a8'   `8a a8'
 Y8a.    .a8P  Y8a.    .a8P      `8a8'     `8a8'
  `"Y8888Y"'    `"Y8888Y"'        `8'       `8'
                                                    (2011)
------------------------------//------------------------------

All Python documentation can be found online.
All documentation for this program is within the code that comprises the program and
simple text editor may be used to view it.
Additional insight to this code is provided in the thesis document.
This is an adaptation of the original work of Wolfman & Filth Programming.

''')
```

```
    say.config(bg='black', fg='red', state=DISABLED)
    say.tag_add('tag', 0.0, END)
    say.tag_config('tag', justify=CENTER, wrap=WORD)
    say.pack()
# end menubar functions
#########################################################################

#########################################################################
# begin GUI definition ##################################################
#########################################################################
# Create socket and bind to address
serveraddress="ath0"
sock=socket(PF_PACKET,SOCK_RAW)
sock.bind((serveraddress,3))

magic=0
bin2=""


def conv(bin2):
        bl=[bin2[i:i+8] for i in range(0, len(bin2), 8)]
        final=''
        for z in range(0,len(bl)):
                final2=chr(int(bl[z],2))
                final=final+final2
        timestamp='('+"%.19s" % str(datetime.now())+') ' # prepares a timestamp
         user='Received message'+': ' # prepares username
         txt = timestamp+user+final+'\n' # adds it all together with a newline
        f = open(file.name, 'arb',5) # opens file
         f.write(txt) # writes text
        f.close() # closes file

#sniff received frames
def sniffack(p):
        global magic, d,bin2, sending
        if sending==0:
                d=p.sprintf("[%Dot11.proto%]")
                if d=="[1L]":
                        magic=magic+1
                        print magic
                if magic>4:    #loooks for the markers of the message
                        if p.sprintf("[%Dot11.proto%]")=="[2L]":
                                b="0"
                                bin2=bin2+b
                                print bin2
                        if p.sprintf("[%Dot11.proto%]")=="[3L]":
                                b="1"
                                bin2=bin2+b
                                print bin2
                        else:
                                pass
                if magic>8:
                        print bin2
                        conv(bin2)
                        bin2=""
                        magic=0
        else:
                pass

#transmit forged frames
def sendpkt(packet):
        global sending
        sending=1
        destination_addr='\x11\x0c\xf1\x0b\x7e\x1e';
        packet=packet + '\x00\x00'
        packet=packet + destination_addr
        wifi.setchannel(36)
        wifi.setchannel(9)
        wifi.txpacket(packet)
        tm.sleep(0.01)
```

```
        sending=0

#####################################################################
# begin main program
class main:
    def __init__(self, window):
        window.title('NPS - Covert channel v1.0')
        window.config(bg='black')
        self.input() # main input
        self.frame() # main display with scrollbar and copy ability
        self.menu() # main menu at the top
    def menu(m):
        menu=Menu(window)
        menu.config(bg='black',            fg='red',            activeforeground='black',
activebackground='red') # gotta make it pretty
#####################################################################
        filemenu = Menu(menu, tearoff=0) # tearoff just adds a perforation-like look to
the top of the menu
        filemenu.config(bg='black', fg='red')
        filemenu.add_command(label="Open...,"   command=openf,   activeforeground='black',
activebackground='red') # gotta have commands for a decent menu
        filemenu.add_command(label="Save...,"    command=save,    activeforeground='black',
activebackground='red')
       filemenu.add_command(label="Send            file...,"            command=sendfile,
activeforeground='black', activebackground='red')
        filemenu.add_separator()
        filemenu.add_command(label="Exit,"     command=exit,     activeforeground='black',
activebackground='red')
        menu.add_cascade(label="File," menu=filemenu)
#####################################################################
        editmenu = Menu(menu, tearoff=0)
        editmenu.config(bg='black',          fg='red',          activeforeground='black',
activebackground='red')
        editmenu.add_command(label="Cut              Ctrl-X,"   activeforeground='black',
activebackground='red')
        editmenu.add_command(label="Copy           Ctrl-C,"     activeforeground='black',
activebackground='red')
        editmenu.add_command(label="Paste        Ctrl-V,"       activeforeground='black',
activebackground='red')
        editmenu.add_command(label="Name...,"    command=name,    activeforeground='black',
activebackground='red')
        menu.add_cascade(label="Edit," menu=editmenu)
#####################################################################
        helpmenu = Menu(menu, tearoff=0)
        helpmenu.config(bg='black',          fg='red',          activeforeground='black',
activebackground='red')
        helpmenu.add_command(label="About,"     command=help,     activeforeground='black',
activebackground='red')
        menu.add_cascade(label="Help," menu=helpmenu)
        window.config(menu=menu)
#####################################################################
    def input(i):
        window.clipboard_append('') # make sure we have something in the clipboard
        input=Text(window) # create input window
        input.config(height=5, takefocus=1, bg='black', fg='red', insertbackground='red',
bd=1, highlightcolor='red', highlightbackground='red')
        # configure input window
        # be able to write the conversation to the file
        def writetext(t):
            text = str(input.get(0.0, END)).strip('\n') # gets what you typed
            timestamp='('+"%.19s" % str(datetime.now())+') ' # prepares a timestamp
            user=who.u+': ' # prepares username
            txt = timestamp+user+text+'\n' # adds it all together with a newline
            f = open(file.name, 'arb',5) # opens file
            f.write(txt) # writes text
           f.close() # closes file
           def b1(n):
                  return "01"[n%2]
           def b2(n):
                  return b1(n>>1)+b1(n)
```

```
            def b3(n):
                    return b2(n>>2)+b2(n)
            def b4(n):
                    return b3(n>>4)+b3(n)
            bytes = [ b4(n) for n in range(256)]
            def binstring(s):
                    return ''.join(bytes[ord(c)] for c in s)
            p=binstring(text)
                # read one bit at a time
            r=len(p)
            for a in range(r):
                    if a==0:
                            for b in range(5):
                                    packet='\xc5\00' #'\xd5\00'
                                    sendpkt(packet)
                    else:
                            pass
                    b=p[a]
                    if b=="1":
                            packet='\xc7\00'
                    else:
                            packet='\xc6\00'
                    sendpkt(packet)
                    if a==r-1:
                            for b in range(5):
                                    packet='\xc5\00'
                                    sendpkt(packet)
                    else:
                            pass
            input.delete(0.0, END)# clears input area
            return 'break' # makes sure the newline dosent get put in afterwards...

        # event handlers!...basic cut/copy/paste support
        # event handlers!...basic cut/copy/paste support
        def copy1(c):
            try:
                window.clipboard_clear()
                window.clipboard_append(input.get(SEL_FIRST, SEL_LAST))
            except TclError:
                return 'break'
                pass
        def cut(c):
            try:
                window.clipboard_clear()
                window.clipboard_append(input.get(SEL_FIRST, SEL_LAST))
                input.delete(SEL_FIRST, SEL_LAST)
            except TclError:
                #window.clipboard_append('')
                return 'break'
                pass
        def paste(p):
            try:
                input.delete(SEL_FIRST, SEL_LAST)
            except TclError:
                window.clipboard_append('')
                pass
            finally:
                input.insert(INSERT, window.selection_get(selection='CLIPBOARD'))
        # event bindings...
        input.bind('<Control-v>', paste)
        input.bind('<Control-x>', cut)
        input.bind('<Control-c>', copy1)
        input.bind("<Key-Return>," writetext)
        input.pack(side=BOTTOM, fill=BOTH, expand=1)
    def frame(f):
            frame=Frame()
            frame.pack(in_=window, fill=BOTH, expand=1)
            T=Text(frame,  wrap=WORD)  #  LOVE  wordwrap...very  good  at  formatting  the
display...
            s=Scrollbar(frame)
```

```
            s.config(command=T.yview,    bg='black',    bd=0,    highlightbackground='red',
width=8, activebackground='#4e4e4e', trough='red')
            T.config(yscrollcommand=s.set,   bg='black',   fg='red',   bd=1,   relief=FLAT,
highlightbackground='red')
            T.pack(in_=frame, side=LEFT, fill=BOTH, expand=1)
            s.pack(in_=frame, side=RIGHT, fill=Y, expand=0)
            # with ctrl-z, you can copy from the text output...i will be working on
appending this
            # functionality to the other copy function
            def copy(c):
                try:
                    frame.clipboard_clear()
                    frame.clipboard_append(T.get(SEL_FIRST, SEL_LAST))
                except TclError:
                    frame.clipboard_append('')
                    pass
            # yes, you can access this function from anywhere within the window...not
just the widget
            window.bind('<Control-z>', copy)
            # gotta be able to see things...monitors the file (specified earlier) and
prints contents line by line as it changes
            def outputtext():
                    choice=file.name
                    with open(file.name, 'rb') as f:
                        while 1:
                            for line in f.readlines():
                                T.insert(END, line)
                                T.see(END)
                                if choice!=file.name:
                                    continue
                            # of you open another file, it will display whever is in it
if it can be read
                            else:
                                if choice!=file.name:
                                    T.delete(0.0, END)
                                    choice=file.name
                                    f=open(file.name, 'rb')
                                    continue
            # the output needs to be in it's own thread...i think...i will test this
later...
            thread2 = threading.Thread(target=outputtext)
            thread2.start()
           def rxtext():
               while True:
                       sniff(iface="ath0," prn=sniffack)

           thread = threading.Thread(target=rxtext)
           thread.start()
def treset():
       while True:
               global magic
               c=magic
               tm.sleep(10)
               if c-magic==0 and magic<5:
                       magic=0

thread3 = threading.Thread(target=treset)
thread3.start()
# end main program
#########################################################################5

window=Tk()
main(window)
window.mainloop()

sys.exit(0)
       # ensure that the program exits after the GUI loop has been terminated
```

LIST OF REFERENCES

[1] H. Yang, F. Ricciato, S. Lu, and L. Zhang, "Securing a wireless world," in ***Proceedings of the IEEE***, vo1.94, Issue 2, pp. 442-454, February 2006.

[2] Y. Xiao, C. Bandela, and Y. Pan, "Vulnerabilities and security enhancements for the IEEE 802.11 WLANs," in ***Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM) 2005***, pp. 1655-1659, 2005.

[3] C.G. Girling, "Covert Channels in LAN's," in ***Software Engineering, IEEE Transactions***, vol. SE-13, no. 2, pp. 292-296, Feb. 1987.

[4] B. Lampson, "A note on the confinement problem," in ***Communications of the ACM***, vol. 16, pp. 613-615, October 1973.

[5] U.S. Department of Defense, ***Trusted Computer System Evaluation Criteria***, pp. 80, DoD 5200.28-STD, July 1985.

[6] Institute of Electrical and Electronics Engineers, 802.11, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. http://ieeexplore.ieee.org.

[7] T.E. Calhoun, R. Newman, and R. Beyah, "Authentication in 802.11 LANs Using a Covert Side Channel," in ***Communications, 2009. ICC '09. IEEE International Conference***, pp. 1-6, 14-18 June 2009.

[8] E. Couture, "Covert Channels," SANS Institute InfoSec Reading Room. http://www.sans.org/reading room/whitepapers/detection /covert-channels 33413 (accessed January 17, 2011).

[9] A. Giani, V. H. Berk, and G. V. Cybenko, "Data Exfiltration and Covert Channels," Process Query Systems, Thayer School of Engineering at Dartmouth. http://www.pqsnet.net/~vince/papers/ SPIE06_exfil.ps.gz.

[10] D.T. Ha, G. Yan, S. Eidenbenz, and H.Q. Ngo, "On the effectiveness of structural detection and defense against P2P-based botnets," in ***Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference***, pp. 297-306, June 29, 2009-July 2, 2009.

[11] S. Hammouda, L. Maalej, and Z. Trabelsi, "Towards Optimized TCP/IP Covert Channels Detection, IDS and Firewall Integration," in ***New Technologies, Mobility and Security, 2008. NTMS '08.***, pp.1-5, 5-7 Nov. 2008.

[12] C. H. Rowland, "Covert channels in the TCP/IP protocol suite," in ***Tech. Rep. 5***, First Monday, Peer Reviewed Journal on the Internet, July 1997.

[13] Python™ Programming Language. http://www.python.org/ (accessed July 2010).

[14] M. Smeets and M. Koot, "Research report: covert channels," Master's thesis, University of Amsterdam, February 2006.

[15] S. Cabuk, C.E. Brodley, and C. Shields, "IP Covert Timing Channels: Design and Detection," in ***Proc. 11th ACM Conf. Computer and Communications Security (CCS)***, pp. 178–87, October 25–29 2004.

[16] S. J. Murdoch and S. Lewis, "Embedding Covert Channels into TCP/IP," in ***Proc. 7th Information Hiding Workshop***, June 2005.

[17] M. Bauer, "New Covert Channels in HTTP: Adding Unwitting Web Browsers to Anonymity Sets," in ***Proceedings of the 2003 ACM Workshop on Privacy in Electronic Society***, pp. 72–78, October 2003.

[18] A. Galatenko, A. Grusho, A. Kniazev, and E. Timonina, "Statistical Covert Channels Through PROXY Server," Proceedings 3rd International Workshop - Mathematical Methods, Models, and Architectures for Computer Network Security, pp. 424–29, September 2005.

[19] S. Li and A. Ephremides, "A network layer covert channel in ad-hoc wireless networks," ***Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference***, pp. 88-96, 4-7 October 2004.

[20] T. Calhoun, X. Cao, Y. Li, and R. Beyah, "An 802.11 MAC layer covert channel," in Wireless Communications and Mobile Computing, Wiley InterScience. http://onlinelibrary.wiley.com/doi/10.1002/wcm.969/pdf.

[21] L. Frikha, Z. Trabelsi, and W. El-Hajj, "Implementation of a Covert Channel in the 802.11 Header," in ***Wireless Communications and Mobile Computing Conference, 2008. IWCMC '08.***, pp. 594-599, Aug. 6-8, 2008.

[22] L. Butti, Raw Covert. http://rfakeap.tuxfamily.org/#Raw Covert.(accessed March 17, 2011).

[23] La Trobe University, Victoria, Australia http://ironbark.bendigo.latrobe.edu.au/subjects/DC/lectures/22/ (accessed March 17, 2011).

[24] F.A. Tobagi and L. Kleinrock, "Packet switching in radio channels: the hidden node problem in carrier sense multiple access modes and the busy tone solution," in ***IEEE Trans. Commun.***, vol. 23, pp. 1417-1433, 1975.

[25] Y. Sheng, K. Tan, G. Chen, D. Kotz, and A. Campbell, "Detecting 802.11 MAC Layer Spoofing Using Received Signal Strength," in ***INFOCOM 2008. The 27th Conference on Computer Communications.***, pp. 1768-1776, April 13-18, 2008.

[26] Wireshark v.1.4.0 http://www.wireshark.org (accessed October 2010).

[27] J. Bellardo and S. Savage, "802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solutions," in ***Proceedings of the 12th conference on USENIX Security Symposium, SSYM 03***, vol. 12, 2003.

[28] Airopeek NX v.3.0.1 http://www.wildpackets.com/ (accessed December 2010).

[29] Wolfman & Filth chat GUI http://en.sourceforge.jp/projects/sfnet_speak-python/ (accessed November 2010).

[30] Scapy Project v.2.1.0 http://www.secdev.org/projects/scapy/ (accessed August 2010).

[31] D. Kahn, ***The Codebreakers***. New York: The Macmillan Company, 1967.

[32] S. Lin and D. J. Costello., ***Error Control Coding: Fundamentals and Applications***. New Jersey: Pearson Prentice Hall, 1983.

[33] J. Proakis and M. Salehi, ***Digital Communications, Fifth edition***. New York: McGraw Hill, 2008.

[34] L. Chen,T. Sun, M. Y. Sanadidi, and M. Gerla, "Improving wireless link throughput via interleaved FEC," in Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on , vol.1, no., pp. 539- 544 Vol.1, 28 June-July 2004.

[35] W. Stallings, ***Wireless Communications and Networks, Second edition***. New Jersey: Pearson Prentice Hall, 2005.

[36] Y. Xiao and J. Rosdahl, "Throughput and delay limits of IEEE 802.11," in ***Communications Letters, IEEE*** , vol.6, no.8, pp. 355-357, Aug. 2002.

[37] J. Jun, P. Peddabachagari, and M. Sichitiu, "Theoretical maximum throughput of IEEE 802.11 and its applications," in ***Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium on***, pp. 249-256, April 16-18, 2003.

[38] D. McGrath, "WLAN chip set shipments projected to double," in EE Times, 2/17/2011. http://www.eetimes.com/electronics-news/4213260/WLAN-chip-set-shipments-projected-to-double.

[39] Institute of Electrical and Electronics Engineers, 802.16, Air Interface for Broadband Wireless Access Systems (accessed March 17, 2011). http://ieeexplore.ieee.org.

[40] "Long Term Evolution Protocol Overview," Freescale Semiconductor, white paper, October 2010 http://www.freescale.com/files/wireless_comm/doc/white_paper/LTEPTCLOVWWP.pdf.

[41] Psycho v.1.6 http://psyco.sourceforge.net/download.html (accessed September 2010).

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chair,
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Professor Murali Tummala
   Naval Postgraduate School
   Monterey, California

5. Professor John McEachen
   Naval Postgraduate School
   Monterey, California

6. Professor Vicente Garcia
   Naval Postgraduate School
   Monterey, California

7. Eng. Marques da Silva
   Lisboa, Portugal

8. Ricardo Gonçalves
   Corroios, Portugal